TURING 图灵程序设计丛书

ELSEVIER 爱思唯尔

Linux and the Unix Philosophy

Linux/Unix 设计思想

[美] Mike Gancarz 著
漆犇 译

- 剖析Linux/Unix制胜之道
- 全新阐释开源哲学
- Jon “maddog” Hall作序并推荐

人民邮电出版社
POSTS & TELECOM PRESS

“Linux和GNU项目的理念表面上是Unix哲学的下一个发展阶段，实际上它只是生生不息的Unix的强势回归。*The Unix Philosophy*第一版中阐述的准则至今仍确切无误，甚至得到更多的佐证。开源除了可以让你清楚地了解到这些编程大师们创建系统的方式，还可以激励你去创建更快、更强大的系统。”

——Jon “maddog” Hall，Linux国际协会执行理事

“Gancarz有效地结合了Unix本身的准则和Linux开发社区中使用的Unix准则，对开源哲学进行了全新的阐释。”

——Henry L. Hall，Wild Open Source首席执行官

Linux and the Unix Philosophy

Linux/Unix设计思想

作为开发Unix多年的专家，Mike Gancarz曾说过：“要想成为计算机的主人，而不是它的奴隶，你就应该使用Linux！”有别于市面上的其他关注如何使用Linux的书籍，这本书讲述的是“Linux的思维方式”，揭示了Linux正是Unix这一无所不能的操作系统的完美实现。到目前为止，没有一本书同时介绍Unix和Linux的设计理念，本书将这两者有效地结合起来，在保留了*The Unix Philosophy*中Unix方面的内容的同时，还探讨了Linux和开源领域的新思想。

- Unix哲学原则的快速参考手册
- 专为读者精心挑选的趣闻轶事，带你身临其境
- 语言风趣幽默，令人尽享阅读的乐趣

本书译自原版*Linux and the Unix Philosophy*，并由Elsevier授权出版

ELSEVIER
爱思唯尔

图灵社区：www.ituring.com.cn
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/程序设计/Linux

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-26692-7

ISBN 978-7-115-26692-7
定价：39.00元

TURING 图灵程序设计丛书

Linux and the Unix Philosophy

Linux/Unix设计思想

[美] Mike Gancarz 著
漆犇 译

人民邮电出版社
北京

图书在版编目（CIP）数据

Linux/Unix设计思想 / (美) 甘卡兹 (Gancarz, M.) 著 ; 漆犇译. — 北京 : 人民邮电出版社, 2012.4
(图灵程序设计丛书)
书名原文: Linux and the Unix Philosophy
ISBN 978-7-115-26692-7

Ⅰ. ①L… Ⅱ. ①甘… ②漆… Ⅲ. ①Linux操作系统—程序设计②UNIX操作系统—程序设计
Ⅳ. ①TP316

中国版本图书馆CIP数据核字(2011)第223243号

内 容 提 要

本书将Linux的开发方式与Unix的原理有效地结合起来，总结出Linux与Unix软件开发中的设计原则。前8章分别介绍了Linux与Unix中9条基本的哲学准则和10条次要准则。第9章和第10章将Unix系统的设计思想与其他系统的设计思想进行了对比。最后介绍了Unix哲学准则在其他领域中的应用。

本书适合所有Linux与Unix操作系统的开发人员阅读，其他系统的开发人员也会从书中阐释的准则中获益。

图灵程序设计丛书

Linux/Unix设计思想

◆ 著 [美] Mike Gancarz
译 漆 犇
责任编辑 杨海玲
执行编辑 丁晓昀

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：12.25
字数：250千字 2012年4月第1版
印数：1－4 000册 2012年4月北京第1次印刷

著作权合同登记号 图字：01-2010-4013号

ISBN 978-7-115-26692-7

定价：39.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版 权 声 明

ISBN:9789812724502

Elsevier (Singapore) Pte Ltd.
3 Killiney Road
#08-01 Winsland House I
Singapore 239519
Tel：(65)6349-0200
Fax：(65)6733-1817

First Published 2012
2012 年初版

每一本书都应该有一句献辞。

本书献给有勇气率先尝试Linux的人们。

继续前进，朋友们！

——MG

译 者 序

“布道者”指的是那些坚定地信仰某一宗教，并且不遗余力地向人们传播此宗教的修道者。本书的作者Mike Gancarz就是这样一位“技术布道者”。他是Linux/Unix最主要的倡导者之一，也是最早开发出X Window System的先驱。他把一些在Unix/Linux社区里口口相传的哲学思想总结提炼，最后集结成这样一本完整的Linux/Unix哲学理论书呈现给所有的读者。是的，我们每一个人都能够站在巨人的肩膀上。

这本书写于2003年，要知道每隔几年计算机世界就会发生沧海桑田般的变化。Google的热潮才刚刚过去，我们现在又有了Facebook、Twitter，还有云计算。“吹尽狂沙始到金”，在这些热潮的背后，Linux/Unix一直都是计算机世界的重要基石，也可以说其哲学教义是这一波又一波网络风潮的动力源泉。而且，正如作者所说的，哲学就是哲学，它不会过时。

这并不是一本讨论技术细节的书，书如其名，它阐明的要点在于“思想”、“道”以及“哲学”。它没有什么高深莫测的知识要点，也没有那些让人觉得晦涩难懂的技术细节，作者通过实例将Linux/Unix的哲学思想娓娓道来。翻阅这本书，你可以体会到Linux/Unix给计算机世界所有人们带来的自由和乐趣。Linux/Unix哲学体系像是一个“宗教”，也代表着一种先进的科学技术文化，其中包含了协作、创新、自由等人类孜孜不倦追求的精神。它的成功不只是因为技术优势，更重要的是它所蕴涵和贯彻的开放与共享精神。

如果你是一名Linux/Unix的高级用户，尽管你大可安心享用你喜欢的这个工具，而不去关心它的基础理论。不过，多了解一些内在机制没有坏处，这是一个锦上添花的过程。如果你是一位门外汉，那么来吧，它会给你打开一扇窗。原来，除了Windows之外，操作系统的世界里也有别样的风景。如果用“武侠”来作一个类比，

这本书就好像是一部教你修炼内功的秘笈，无论新手老手，修炼基本内功都是一件必须持之以恒甚至可以毕生研习的事情，而同时我们也要知道，有时候优秀程序员和普通程序员水平差距的关键也正在于此。

本书的组织结构在作者Mike Gancarz自己撰写的前言中已经做了归纳总结，此处不再赘言。

此外，我在翻译本书的过程中，得到了很多朋友的帮助，他们是：刘园园、向梓鑫、冯海涛、常亚平和冯乐宇。我还要感谢图灵的编辑杨海玲、何建辉等，谢谢他们的信任与大力支持。此外，在图灵社区书稿试读活动中，李琳骁、周兵、臧秀涛、张伸等网友也提供了很多宝贵的意见，我也要向他们表示衷心的谢意。

漆犇 于上海

序

对我来说，1969年发生了三件大事。

那时候我还只是美国东部一所知名大学的学生，无意中我发现了一项引人入胜的新业务——编写计算机软件。当时的大街小巷里根本就没有什么计算机商店，更没地方去购买现成的包装好的现成软件。事实上，那会儿普通计算机的内存还不到4000个“字”（word），处理器每分钟只能处理几百万条指令（而现在的处理器每秒就能处理几十亿条指令），并且，普通计算机的售价甚至高达几十万美元。在当时，开发软件的宗旨就是充分利用机器的功能。那时候，编写软件都是从零开始，由客户确定并提供输入内容，同时也要明确他们预期的输出。随后软件团队（无论是公司内部的团队还是外部的顾问）负责编写软件来完成该项转换工作。如果软件不能工作，那编程人员就得不到报酬。显然，就像任何量身定做的项目一样，获取软件的代价不菲，硬件产品也是如此。

幸运的是，DEC公司用户协会（DECUS）有一个程序库，人们可以将自己编写的软件贡献出来供他人使用。程序作者会将他们的代码提交到DECUS程序库里，然后程序库管理人员会制作一份程序目录，复制和邮寄目录的费用都是由那些要求得到程序副本的人们来承担。软件是免费的，但它们的分发（以及印刷目录等的费用）却需要费用。

我通过DECUS得到了很多免费软件。这简直就是太棒了，因为当时我只是一个穷学生，没钱去购买定制的软件。我必须在软件和啤酒之间做一个抉择，要知道1969年用来购买文本编辑器软件的5美元可以买10扎啤酒呢。如果要试用并购买“商业”软件的话，那我就该喝西北风去了。

为什么这些人会把自己编写的软件贡献出来呢？他们之所以编写软件，是因为他们在工作或研究中有使用软件的需要。他们（大方地）认为，或许其他人也能从

中受益。他们还（合理地）希望软件的使用者可以帮他们改善这些软件，或者至少给他们提一些改进意见。

1969年发生的第二个重大事件就是，Ken Thompson、Dennis Ritchie和其他一些在新泽西州AT&T公司（美国电话电报公司）贝尔实验室的研究人员一起开始编写一款操作系统，最终这成为众人皆知的“UNIX”。在编写该操作系统的过程中，他们形成了一套理论，那就是操作系统必须由一些小的可重复使用的模块组成，而不是一组大型的独立程序。此外，他们还为操作系统开发出具有“可移植性”的架构，完全不必理会计算机的主CPU指令集，它就能运行在从最小的嵌入式设备到最大的超级计算机等各种设备上。最终，这个操作系统辗转流传到了大学里，然后又被传播到美国各大公司。在这些地方，它主要是在服务器上运行。

第三个重大事件在未来很大程度上改变了我和其他几百万人的生活。但当时，除了身在芬兰赫尔辛基一对自豪的父母外，没有人注意到这件事情，那就是Linus Torvalds的诞生，他就是未来的Linux之父。

接下来的十年，计算机科学稳步发展。各大公司都开始投资开发新的软件和技术。虽然每家公司对计算机应用的研究日趋深入，但他们仍遵循着一个原则，那就是购买量身定做的软件。当然，按照今天的标准来说，他们所使用的计算机容量出奇地小、速度出奇地缓慢、代价出奇地昂贵。然而，AT&T公司的Unix开发者和给予过他们帮助的一些大学一直在创建一种鼓励软件重用的操作系统，同时，为了避免重写代码、重复创造，这些人并没有一味追求代码的高效运行。大学和计算机科学的研究者们乐于获取操作系统的源代码，以便他们能够共同完善其功能。那真是一个美好的年代。

然后在20世纪80年代初又发生了三件大事。

第一件事是Unix的商业化。这时候，小型机的成本已经下降，而编程和用户培训这一块的成本开始超出了基本硬件成本。Sun公司开创了一类新市场，其中的客户都要求拥有一个可以运行在多种处理器之上的“开放式系统”，而不是当时像MVS、MPE、VMS以及其他的商业操作系统那样的“专有”系统。一些公司如DEC、IBM和惠普都开始考虑为自己创造一个“商业化”Unix版本。为了保护他们的投资，并从AT&T公司那里获得更低的专利费用，他们只采用二进制文件的形式来发布自己的系统。因此，Unix系统源代码的价格变得非常昂贵，几乎没人能负担得起。通过共享方式来改进Unix的举措便被暂时搁置。

第二件事便是微处理器的出现。最初，它创造了一种经由BBS、杂志、计算机俱乐部来共享软件的氛围。但随后新的概念又产生了，那就是“带塑料薄膜包装的”软件。这些软件是针对那些 “商品化”处理器架构而编写的，比如英特尔或摩托罗拉的产品；那时硬件的生产量级也只是上百或数千而已。最早的商业化操作系统是CP/M，后来变成微软的MS-DOS和苹果的操作系统，商业软件产品的数量持续增长。我仍然记得自己第一次走进电脑商店时的情景，我看到了很多来自不同厂商、有着不同硬件架构的计算机摆在货架上，可软件产品却只那么三四个（包括一套文字处理软件、一款电子表格、某种“调制解调器软件”产品）。当然，这些软件产品都不附带其源代码。如果在有了这些软件及其相关文档后，还是无法让它们运行的话，那你就无计可施。渐渐地，以公开协作方式编写软件的方法被一些商业做法所取代，这些做法包括采用二进制文件发布产品、最终用户许可证（告诉你该如何使用你所购买的软件）、软件专利和永久版权。

幸运的是，在那些日子里，每家公司的客户数量都不太大。只要致电各公司的服务热线就可以得到技术支持。不过如果想要以这些公司开发的软件为基础添加新功能，那是一项几乎不可能完成的任务。

第三件事实际上是前两个事件的结果。在麻省理工学院的一个小办公室里，一位名叫Richard Stallman的研究员决定解密Unix和其他软件的源代码。他讨厌这种愈演愈烈的软件闭源趋势，并决定于1984年启动一个项目来编写一个永远开放源代码的完整操作系统。当然，源代码自由发布有其副作用，也就是对那些愿意获取源代码来编译自己操作系统的人们来说，后续开发出的软件也应该永远是免费的。他把这个项目称为“GNU”，意为“GNU不是Unix”（GNU is Not Unix），以表达他得不到Unix源代码的愤懑之情。

时光依然在流逝。微软成为了操作系统产业的主导。与此同时，其他系统供应商也发布了不同版本的Unix，打着所谓“创新”的名号，其中大部分版本都互不兼容。整个市场充斥着各种各样僵化死板的商业软件，每个精心包装的程序都是为某个需要独特解决方案的商品市场而编写的。

接下来，在1990年前后，再一次发生了三件大事。

第一件事情是一小群Unix专业人员围坐在一张桌子旁，比较着不同市场中不同类型的软件，其中一个人发问道（这个人就是我）：

“你为什么喜欢Unix？”

在座的大多数人一开始都不知道该怎么回答，这个问题久久地萦绕在会议桌上方，然后我们每个人开始给出自己热爱并忠于这个操作系统的理由。“代码的重用”，“高效、精心编写的实用程序”，“简单，但是优雅”，诸如此类理由被人们提出来并得到了详细阐述。我们当中没有一个人注意到，在大家提出这些想法的同时，有一个人正在做详细的记录。后来，这些想法成为了本书第一版的核心思想，而且那也是市面上第一次出现这样一本关于Unix设计思想的“哲学书”。

为什么大家需要一本关于“Unix设计思想”的书呢？这是为了帮助Unix初学者了解该系统的真正威力和优雅设计底蕴。向他们展示在编写程序时使用Unix下的工具和结构可以节省大量人力物力。Unix还可以帮助他们扩展计算机系统的功能，而不是帮倒忙。可以这么说，Unix让人们能够站在前人的肩膀之上。

第二件事情是，1990年GNU项目已完成大半。它们包括命令解释器、系统工具、编译器、各种库文件，等等。这些GNU软件与其他一些免费软件如Sendmail、BIND和X Window System结合在一起，只是唯独少了操作系统的核心，也就是被称为“内核”的那一部分。

内核是系统的大脑，它控制程序运行的时间、程序能够得到哪些内存、程序可以打开什么文件，等等。它被留在最后实现的原因是因为内核每天都在发生变化，随时都在改进。一个没能提供任何工具或是命令解释器的内核没有任何用处。多年以来，让所有的工具能为其他操作系统和其他内核所用是一件非常有意义的事。

第三件事情就是，1990年的12月Linus Torvalds已经成长为一名就读于芬兰赫尔辛基大学的学生，而且他刚刚得到了一台最新的英特尔386电脑。Linus认识到，当时微软的操作系统并没有充分利用386的能力。于是，他决定自行编写一个内核，将它和其他免费的软件结合在一起，创建一个完整的操作系统。待到事后他才确定，这个内核应该遵循GNU项目的通用公共许可证（General Public License，GPL）。他在互联网上的一些新闻组发布了关于这个内核项目的消息，并开始了工作。

1994年4月我了解到Kurt Reisler，也就是DECUS的Unix SIG（Special Interest Group，特别兴趣小组）的主席，试图筹集资金让一名程序员来到美国与DECUS的SIG小组讨论他正在开发的项目。最后，我请求DEC的管理层提供了这笔资金，因为我相信Kurt一贯秉持的远见卓识和敏锐的洞察力。1994年5月，我参加了DECUS在新奥尔良举办的这次活动，并见到了这位程序员，他就是Linus Torvalds。从第一眼看到他的操作系统开始，我的生活就此改变。

在那之前的几年里，我一直都在大力主张重新编写这个我热爱了多年的Unix操作系统，不过其实现方式必须能够鼓励人们在他人的工作基础上去阅读、修改和完善源代码。事实上Unix的成长远远超出了大多数人对它的基本认识，因为自由软件运动已经囊括了数据库、多媒体软件、商业软件以及其他对人们有价值的软件。

而且，软件的生产潮流又一次改变了。硬件工艺越来越精细，价格却越来越便宜，互联网上的协作开发也越来越容易，软件信息传播的速度如此之快，以至于百转千回之后，主流的开发组终于能够走到一起来开发那些可以帮助他们解决自己问题的软件。软件再也不需要由那些自诩为“德鲁伊”（druid）[①]的人来开发，他们坐在高耸入云的“大教堂”里使用昂贵机器创建程序。现在，每个人都可以发出“代码就在这里”的呼声。这些人可能就坐在自己的家中或教室里，他们都能为计算机科学的发展作出自己的杰出贡献，无论是身在美国、巴西、中国还是芬兰的赫尔辛基。成千上万的项目开始启动，不计其数的程序员在为它们卖力工作，这种态势不断扩大，发展速度也日渐迅猛。

未来的计算机编程世界里，单凭程序员小团体不大可能可以完成百分百满足所有人需求的软件。相反，网络上会出现一大批以开源形式存在的项目，它们的开发人员盼望着业内顾问和增值的经销商能够将这些软件加以改进，并定制出针对客户精确需求的解决方案。

Linux和GNU项目的理念看似是Unix哲学的“下一个发展阶段”，其实它不过是Unix的强势回归。*The UNIX Philosophy*[②]一书中阐明的准则直到今天依然适用，甚至其重要性变得益发明显。源代码的可用性还让人们清楚地了解到这些编程大师们是如何创建他们的系统，而且也能激励你自己去编写功能更强、运行速度更快的代码。

借助巨人的肩膀，愿你能攀得更高。

祝学习愉快！

Jon“maddog”Hall

Linux国际协会（Linux Internopational）执行理事

---

① 德鲁伊属于欧洲“凯尔特”人中的特权阶级，是部落的支配者、王室顾问、神的代言人，有着至尊地位。——译者注

② *The Unix Philosophy*，作者Mike Gancarz，Digital Press，1995年出版。

致　谢

一本书的诞生就像是孩子的出世：每个都是不同的。我第一本书*The UNIX Philosophy*的撰写和出版都相当顺利。但如果预先知道在写第二本书的时候会面临重重困难，我都不大可能会签下这份出版合同。我当时真应该三思而后行。

在经历了发生在写作和出版本书过程中的个人波折之后，我承认这本书就像是一种个人和哲学两方面的宣泄。所以，伟大全能的上帝功不可没，他再一次引领我渡过难关。他是我优先并重点感谢的对象，上帝创造了我，使我得以完成这个使命。

我要感谢巴特沃斯-海纳曼（Butterworth-Heinemann）出版集团旗下数字出版社（Digital Press）的策划编辑Pam Chester，是她不遗余力地劝说我创作这个新版本，并且每当出现问题，她都不断地鼓励我。我还要感谢她的老板Theron Shreve，是他的耐心让本书最终付梓。

我很幸运，有一群杰出的开发人员帮我审阅这本书。我要谢谢Jim Lieb、Scott Morris、Rob Lembree和Jes Sorensen提出的精彩见解。他们的意见促使我从不同的视角去看待事物，并且深化了本书中不少有趣的内容。他们还在最后阶段给了我不少鼓励。

我尤其要感谢的是Jon“maddog”Hall，不仅因为他给本书写了一篇精彩的序言，更因为他倡导Unix和Linux的巨大努力，这给我们所有人塑造了一个典范。几乎没有人能够像他这样坚持不懈，把Unix的理念传达给这么多人。

我还要感谢数字出版社的前任出版人Phil Sutherland，我们就本书最初的想法进行了一系列的电子邮件交流。尽管最后本书并没有以我们两个之前设想的方式完成，但他劝服我创作本书的努力并没有白费。这件事情就是需要花费更长一些时间。

家里有一个作者会让每一位家庭成员体会到与作家一起生活的滋味。谢谢你，Sarah，你为我打了那么多字，这个活儿对你来说一定很无聊。还有Adam，写书的时间本是应该用来与你玩彩弹射击的，对你作出的牺牲我表示深表谢意。最重要的是，我要感谢我的妻子Viv始终如一地支持整本书的创作，我欠你的“老公该做的事情”的清单现在肯定都有一千多米长了。

引　言

*The UNIX Philosophy*出版了几年之后，数字出版社（Digital Press）的前出版人Phil Sutherland再次和我联系，约我创作一本关于Linux的书。他认为Linux很快就会成为计算机世界的重要角色，并相信市场需要一本“Linux哲学”之类主题的书。我们通了一年多电子邮件，就这个主题展开了深入探讨。市场的需求变得日益明朗，可写的东西也很多，只是我总觉得还没有找到感觉。

Phil督促我写一本Linux版的*The UNIX Philosophy*。从市场营销的角度来说，很有道理；然而，我却观察到Linux社区有一些不同寻常的迹象，Unix的世界也正勃发出一股新鲜活力。很明显，老牌Unix信徒已经成功地将有关“Unix思维方式”的思想灌输到了新一代人的头脑里，这些人不但包括充满激情的黑客，还有其他热衷于在自己机器上探寻Unix奥秘的狂热爱好者。是的，Linux就是Unix，只不过它已不再是上一个时代的Unix。

我也曾考虑过写一本名为“The Linux Manifesto”（Linux宣言）的书，在里面描述这种创新精神，可最终我清楚地意识到，“开源”才是Linux的精髓所在。因此，为了更好地契合主题，我得把“Linux宣言”更名为“开源宣言”。但是，开源社区领袖Eric Raymond早就创作过一本关于“我们为什么要这样做”的经典大作——*The Cathedral and the Bazaar*[①]。我并不想东施效颦，于是，便只能暂时搁置这个想法。

不过，虽然最初的想法没有形成作品，好想法却总会改头换面，以一种不同的形式再度出现。两年前，数字出版社的Pam Chester向我指出，*The UNIX Philosophy*中的一些信息已经过时，问我是否有兴趣做一下修订。可是，哲学怎么能修订呢？

---

① *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*（大教堂与集市：一个偶然革命引发的关于Linux和开放源码之深思），作者Eric S. Raymond, O'Reilly & Associates，2001年出版。

如果它就是真理，那么，不管是一个月、一年，还是几百年以后，它始终都是真理。

后来，我终于想明白了。*The UNIX Philosophy*描述了一种Unix的思维方式，它其实就是“第一代系统”；但现在，它正在演变成“第二代系统”。而且就本质来说，相比“第一代系统”，这“第二代系统”是一个更全面、更成熟、更有价值的版本。虽然*The UNIX Philosophy*讲述了其最基本的原理，但是本书会将原来的概念提炼升华，而且还探索到新的领域。当然，它也仍然保留了原来的哲学信条——因为，真理永远是真理。

因此，*The UNIX Philosophy*的读者们会发现，本书有很多他们已经谙熟于心的思想。这是因为，Unix哲学的基本准则并没有发生改变。不过，我还是重新审阅了每一个章节，仔细思量该如何才能恰如其分地在Linux语境下阐明这些准则。从某种意义上来说，这本书是*The UNIX Philosophy*的修订稿，但它也探索了新的领域。本书的许多技术审稿人在反馈中也都谈到了这一点。这本书给了他们有别于前版的全新视角去审视Unix。一方面，它描述了Unix哲学对Unix程序员编写操作系统所产生的影响；另一方面，它还提出了一个更为大胆的命题，那就是，展示这套哲学理念对计算机世界其他领域的影响，其影响甚至还涉及计算机世界之外的天地。

Linux和开源软件的出现，使得软件世界面临洗牌。Unix哲学正在成为计算机行业的主导思想。很多人都采用了它的准则。这些思想造成了整个行业的巨大动荡，原来的偶像纷纷被抛弃，人们开始探寻一种不同的方式，以个人角度去利用计算技术。Linux正好填补了这一空白，为主流电脑用户提供了使用Unix的第一手体验。越来越多人开始使用Linux，他们见证并亲历着这种全新的工作方式。另一方面，Unix的信徒也发现，个人计算技术又重新回到了他们熟悉的领域。

其实，Linux哲学就是Unix哲学的加强版，这是我要着重强调的。当然这与一些人的观念冲突了，他们宁愿相信“Linux不是Unix”。不过我由衷地承认，作为一种现象，Linux确实与Unix表现出来的特性颇为不同。在软件开发过程中，Linux社区有意克服了专有软件的开发模式，但Unix社区一直没有清晰地意识到其实它也应该这么做。

Linux社区较为精明，也更了解市场。它意识到，Linux想要成功，就必须在每一轮竞争中都出其不意，想别人所不能想，做别人所不能做。迄今为止，它的表现非常出色。据传，Linux开发人员已经超过一百万人。他们誓言要去“攻占”其他

操作系统开发人员的地盘，在每一个计算机重要领域里都取得杰出成就——不管是安装使用的便捷性、图形用户界面、硬件兼容性、可靠性、安全性、整体性能、网络开发、数据库，还是游戏，等等。

经过整个Linux开发社区的不懈努力，Linux已经成为计算机行业中不可或缺的一员，任何人都无法否认这个事实。

谁会是本书的受益者

最早的时候，只有系统程序员才会对Unix的资料感兴趣。今天，Linux用户和开发人员组成了一个更加多元化的社区，容纳着各种个人爱好和专业兴趣。在Linux成为计算机世界主流的同时，那些从未听说过Unix的人们就已经见过或用过Linux，甚至拥有了自己的Linux系统。在本书中，我避免谈及那些底层的技术细节，力求给这些人提供Unix、Linux和开源软件中极富启发性的观点。这些未曾深入研究过其哲学教义的人们会发现，他们无需陷在命令参数、编程接口等技术细节里，就可以通过本书了解到Unix的思想，以及这些思想在Linux中的实现。

Linux开发人员可以从这本书中领会到Unix的奠基性思想，同时，也会从介绍Linux是如何接受并发扬这些思想的内容中获益。他们会认识到，尽管Linux拥有出色的图形用户界面环境，但是其内在价值是源于它背后的哲学理念。想要成为一个真正的Linux “超级用户”，就必须去了解：为什么Linux基于文本的理念和工具会有着压倒性的竞争优势？

“老牌”Unix程序员可以考虑读一读本书，这会让他们明白，为什么Unix没有消亡，而且为什么它永远都不会消亡。在Linux的世界中，Unix如凤凰涅槃，以功能更强的新形式重生。这本书还会告诉他们，在新的千年里，Linux将如何以全新的、激动人心的方式来满足计算需求，同时依然固守着初始的Unix哲学准则。

在当前快节奏、竞争日趋激烈的开发环境下，一些经验丰富的开发人员经常会感到，迫于巨大压力他们往往会背离优秀软件的设计原则。然而，很多情况，公司只是为了追赶最新潮流而使用新的开发流程，而不是为了确保他们所选择的方法有足够的事实根据。那么，当设计中有看起来无法解决的问题，或是解决方案的前提条件令人心存疑虑时，这本书可以帮助你进行理性反思。

今天，商业世界的IT主管们对Linux操作系统也有所耳闻。出于节约成本的目的，许多人正在考虑将Linux引入到他们的环境中。他们通常会采取这样的策略：在一些不那么关键的应用上，使用相对便宜的英特尔或AMD处理器的Linux服务器来取代老旧的Solaris和HP-UX平台。随着IBM公司和甲骨文公司正逐步加强他们的Linux产品，相较前几年而言，会有更多人接受将主要服务器切换到Linux平台的方案。

然而，迄今为止，IT主管们一直不敢将自己的桌面用户迁移到Linux平台。这本书会让他们意识到，Linux不仅是一个可行选项，实际上它还是更好的替代方案。他们会发现，相比同类型的Windows产品，Linux和开源软件蕴含的哲学实际上使得台式机更易于维护，成本更低，也更安全。这些哲学理念不仅让Linux成为一款理想的服务器，也会在桌面产品上发扬光大。所以，在人们不以为然地回应“是的，但是……”之前，我只想说一句话：来读一读本书吧！

最后，从事抽象思维工作的人也能从Unix哲学的学习过程中发现价值。可以这么说，很多软件开发的技巧同样适用于其他行业。作家、平面设计师、教师和演讲家可能会发现，“快速建立原型”和“巧用杠杆效应”在他们各自的领域也能发挥作用。而且，随着大量价廉物美甚至是免费的Linux应用程序的出现，人们可以尽情享用那些在过去因太昂贵而无法使用的工具。

章节概述

第1章——Unix哲学：集思广益的智慧。这一章探讨了Unix哲学的发展史，以及它是如何成型的。我还概要介绍了一些Unix哲学的基本准则，作为对后续章节中长篇论述的序曲。

第2章——人类的一小步。这一章说明了为什么以小模块来构建大型系统是最佳的方案。它详细探讨了无论是在软件系统还是现实世界中，小型模块都能较好地通过接口组装在一起。这一章的最后部分阐释了“让程序只做好一件事”的重要性。

第3章——快速建立原型的乐趣和好处。这一章强调，想要设计一款成功产品，就要尽早建立原型。我们将讨论“人类构建的三个系统”这一主题，这是所有软件都要历经的开发阶段。如果想要构建三类系统中最完善、最稳健的那“第三个系统”，那“快速建立原型”就是最快也是最佳的途径。

第4章——可移植性的优先权。这一章以不同视角来审视软件的可移植性。这里会强调，设计过程中软件开发人员必须在效率和可移植性之间作出取舍。本章以雅达利公司的视频计算机系统（Atari VCS）为范例来探讨高效率和有限移植性。这一章还强调了一个非常重要却往往被人们忽视的目标，也就是数据的可移植性。此外，这一章还对典型Unix用户的工具集合做了个案研究，并借此完美地验证了一个事实，那就是可移植性能够延长软件的使用寿命。

第5章——软件的杠杆效应。这一章讨论了“软件杠杆效应”，也就是可重用组件带来的巨大影响。我们可以看到，怎样使用shell脚本来获取超强的软件杠杆作用。

第6章——交互式程序的高风险。这一章开篇就定义了什么是“强制性的用户界面”（Captive User Interface），并建议开发者们少用这类界面，而是集中精力去让程序之间能够更好地交互。这一章表达了一个思想：所有的程序都是过滤器。这一章末尾将讨论“在UNIX环境中的过滤器”。

第7章——更多Unix哲学：十条小准则。这一章列举了Unix开发人员普遍遵循的一些概念，但它们没被当做是Unix哲学的主要内容。由于这一章探讨的是更为底层的概念，对技术不那么感兴趣的读者可以跳过这一章。然而，纯粹的Unix“信徒”很可能会发现这一章的内容相当有趣。

第8章——让Unix只做好一件事。这一章通过Unix邮件处理程序MH来说明该如何构建良好的Unix应用程序。章节末尾还对Unix的哲学准则加以总结，展示怎样才能将各个模块集结在一起以获得强大的功能。

第9章——Unix和其他操作系统的哲学。这一章将Linux和Unix的理念与其他几个操作系统的哲学进行了对比，从而强调了Unix的独特性。

第10章——拨开层层迷雾：Linux与Windows的比较。这一章开篇便对Linux和微软的Windows做了一个深层比较。通过事实来对比“基于文本的系统”与“基于图形用户界面的系统”的情况。这一章还归纳总结了Windows系统中哪些方面遵循了Unix哲学，而哪些又是完全与之背离的。

第11章——大教堂？多怪异。这一章探讨了Linux社区推广的开源软件（Open Source Software，OSS）概念是如何契合Unix哲学理念的。当开放源码软件在各个层次均采用“Unix的思维方式”时，它的传播效果最佳。

第12章——Unix的美丽新世界。这一章论述了新技术领域采用Unix哲学准则的情况与前景。它着重于几大关键点和一些不那么关键的技术，并借此来说明Unix的想法不再是只为Unix开发人员所用。对于这一章，本书技术审稿人最常见的评论就是："我从没想过可以用那种方法。"读一读这一章，它会开阔你的眼界。

经过以上对本书的概述，你可能已经看出来，我在文中尽量避免涉及过多的技术细节，因为各个Linux发行版中细节都有所不同。我在书中强调，Unix哲学其实是一种设计方法论，它偏重于更高层次上的通用方法，而不是某些具体的细节。

我努力保持本书的文字轻松易懂。虽然可能有人喜欢阅读那些冷冰冰、不带感情色彩的大部头技术资料，但我们大多数人还是更喜欢兼具娱乐性和知识性的文字，也就是"信息娱乐"（infotainment）。这种理念也暗合了Linux和Unix的文化。在Unix社区及后起的Linux社区中，一直都有保持幽默机智的传统。此感觉也体现在各个方面，从最早贝尔实验室的Unix文件到Linux开发人员经常光顾的网站。也许，这就是为什么Linux开发人员会如此享受他们工作的一个理由。

当然，你也不要被这些"浮华文字"的外表所蒙蔽。我们探讨的是颇为严肃的事物。遵守这些准则，就能从软件中获得价值；忽视这些准则，就会失去这些价值。恰当运用Unix哲学中蕴含的原理有助于你编写出非常成功的软件产品。如果在开发过程中背离这些原则，则往往会让开发者们错过市场的良机。

听过我讲座的学生们提到，一开始他们对这些思想还只是一知半解，不过，你会时不时回想起它们。强大的思想都有这样的魔力。如果你从来都没有接触过Unix哲学，那么就请准备好迎接一次有趣的旅程吧！

本书还大致介绍了*The UNIX Philosophy*一书。这样做有两个原因。首先，人们很喜欢在互联网上转发里面的准则，使其成了不朽传奇，所以，就算是没买过那本书的人也能从本书中受益。第二个原因就是，我在那本书的最后一段作出了预测："Unix必将成为一个风靡全世界的操作系统，只是时间早晚的问题。"而现在，我们见证到了，源自Unix系统的Linux操作系统正在通过每天每台机器中的每个应用实现着这个预言。

The Unix Philosophy 简介

操作系统是一个生机勃勃的软件实体。它就像是计算机的灵魂和神经系统，赋予每一个电子器件和硅片元件生动的个性。它给计算机带来了生命。

究其本质而言，操作系统体现着创造者的哲学思想。苹果的Mac / OS系统通过非常直观和高度面向对象的用户界面来宣称："它就在那里，在你的面前。"作为个人电脑革命中无可争议的领军产品，微软的MS-DOS系统试图带给桌面用户一种"主流系统的味道"。此外，美国老牌电脑公司DEC的OpenVMS系统则认为用户畏惧这种有电子思维的机器，所以只提供为数不多的几个强大选项帮助用户完成某个任务。

在起步阶段，Unix操作系统的创作者就采取了一种颇为激进的理念：他们假定自己的软件用户从一开始就能够熟练操作计算机。于是，整个Unix哲学都围绕着"用户知道自己在做什么"的思想而展开。其他操作系统的设计者总是煞费苦心地去迎合各层次用户，从新手到专家；而Unix的设计者则采取了比较冷淡的态度："如果你不能理解，那你就不属于这个世界"。

正是因为这样的态度，Unix在初期无法得到广泛接受。它只局限于学术研究界的小角落里，因为它给象牙塔里的人们带来了一种令人陶醉的神秘气息。("是的，Unix文件系统的层次和万物的自然规律仿佛有共通之处，处处体现着造物者的神奇。")此外，"技术偏执狂"也喜欢随意摆弄Unix，比起以往的任何系统，Unix给出的自由空间更多。

令人扼腕的是，商业世界却未能看到它的价值所在。Unix渐渐成为黑客的玩具，处处透着神秘。几乎没有盈利性企业敢去冒着投资风险，采用一个出自研究实验室，由大学资助而且只能靠购买方自行作技术支持的操作系统。因此，在其诞生后的15年内，Unix一直如世外高手般默默无闻。

然后在20世纪80年代初期，惊天逆转开始了。人们纷纷开始流传一个传说：比起人们现在使用的任何操作系统，有一个操作系统更加灵活、更易移植，而且还拥有更强大的性能。此外，它还能以非常低廉的成本运行在任何机器上。

关于Unix的这些消息，听起来似乎有点儿好得令人难以置信，但历史总是能够证明一切。不管何时，当某个彻底改变大家世界观的激进想法出现时，我们会本能地去怪罪那些带来新思想的信使。计算机主流世界的任何人都可以看到，这些"Unix的狂热分子"讨论的并非技术改良，而是技术革命。

正当Unix开始渗透到计算机世界时，很显然许多企业官僚都厌恶这种革命性的思想。他们更喜欢由个人电脑和大型机构成的井然有序世界。他们坚持认为，掌握

那些他们绞尽脑汁才学得会的简单命令来应付日常工作安稳地保住自己的饭碗就够了。Unix成为洪水猛兽，并不是因为它具有邪恶本质，而是因为它威胁到了现状。

多年以来，Unix先驱都过着默默无闻的生活。找不到人来支持他们的激进观点。即使一些有同情心的人士愿意聆听Unix倡导者的长篇大论，他们通常的反应也只是“Unix是不错，但如果想真正做些工作，你应该使用____。”（请在空白处填写你最喜爱的主流操作系统的名字。）但是，操作系统的哲学有点儿类似于信仰。当一个人自以为找到真理时，他是不会轻易放弃的。于是，Unix信徒们只能继续勇往直前，顽强地坚守着他们的信仰，并相信总有一天世界会改变，软件的理想国终将来临。

商业界忙于建造阻碍Unix发展的壁垒，而学术界却张开双臂欢迎它的到来。那些伴随着彩电、微波炉、视频游戏成长起来的年轻一代步入大学校园，而这些大学已经可以几乎不费分文地获得到Unix的发行版光盘。这些年轻人纯洁无暇的内心犹如洁白画布，教授们更愿意在这样的白纸上绘制计算机世界非主流的蓝图。

接下来的故事尽人皆知。

如今，就算是在过去不可想象的领域，Unix也已获得了人们的广泛认可。在学术界，它更无可争议地成为首选系统，其在军事和商业领域的应用也一天天地扩大。

多年以来，我不断告诉人们，Unix迟早会成为风靡全世界的操作系统。我的预言并没有失手。然而，很讽刺的是，这一款操作系统却不能再叫做“Unix”了。因为有些企业早已意识到这个名字蕴含的商业价值，并委托他们的律师抢先一步将其注册为自己的商标。因此，未来人们设计出的界面、提出的标准，以及诸多的应用只能被命名为“开放式系统”。不过大家可以放心，很显然，Unix哲学还将会是它们背后的驱动力。

目　录

第1章 Unix哲学：集思广益的智慧

这个世纪的哲学会成为下一个世纪的常识。

——中国幸运饼干[①]

许多人都将发明Unix操作系统的殊荣授予AT&T公司的Ken Thompson，从某种意义上来说，他们是对的。1969年在新泽西州美利山AT&T公司的贝尔实验室，Thompson编写出了Unix的第一个版本。它作为Space Travel程序的平台运行在Digital PDP-7小型机上。此前，Space Travel程序运行在由麻省理工学院开发的Multics系统上。

Unix的开发基于Multics系统，后者属于最早的一批分时操作系统。在Multics开发之前，大多数计算机操作系统都运行在批处理模式下，这迫使程序员们要去编辑大堆的打孔卡片或纸带。在那些日子里，编程是一个耗时费力的过程。当时有一句流行语是："上帝帮帮那些打翻了打孔卡片盒的傻瓜吧。"干过卡片机编程的人都懂。

Thompson借鉴了Multics的许多特性，并将它们融入到早期的Unix版本，其中最主要的特点就是分时处理。如果没有这种特性，那些在当前Unix系统或是其他操作系统上被人们视作理所当然的大部分功能，就会失去它们真正的力量。

Thompson的开发工作从借鉴Multics的想法入手，对于Unix开发人员而言，这样的套路可谓是驾轻就熟：良好的程序员写出优秀的软件，优秀的程序员"窃取"

① 在美国的中餐馆吃完饭后，服务生一般会送上一小碟幸运饼干（Fortune Cookie），外表金黄，呈菱角状，脆脆的没有什么特殊的味道。这个饼干可以用手掰开，里面有一张纸条，写着一些关于运势的签语。——编者注

优秀的软件。当然，我们并没有暗示Thompson是一个小偷。但正是他这种在某些方面避免NIH（Not Invented Here，非我发明）综合征的意愿和基于别人的成果添加颇具创造性价值的做法，大力推动了这一款或许是历史上最精巧操作系统的出台。我们还将在后面探讨“窃取”软件的意义。现在只需要记住，将一个想法与人共享就如同一个大脑里有了两个想法。

1.1　NIH综合征

软件开发人员经常会受到NIH综合征的影响。在查看别人编写的软件解决方案时，他认为自己完全可以做得更好。也许他真的能更为痛快利落地完成这项工作，但他并不知道别的开发人员当时面临的限制条件。他们可能迫于时间或预算的压力，于是，只能集中精力处理这个解决方案中的某些特定部分。

NIH综合征的特点就是人们会为了证明自己能够提供更加卓越的解决方案而放弃其他开发人员已经完成的工作。这种狂妄自大的行径说明此人并无兴趣去维护他人竭尽全力提供的最佳工作成果，也不想以此为基础去挑战新的高度。这不仅是个自私自利的做法，还浪费了大量宝贵时间，而这些时间其实完全可以用来提供其他解决方案。更糟的是，新的解决方案有时候只是稍作了一些改进，或根本没有本质区别，从而使得这个问题变得更糟。

偶尔，新的解决方案确实会更好，这只是因为开发人员早已了解前人做过的工作，因此他们可以“取其精华，去其糟粕”，改善此解决方案。这是对前人工作的加强和延伸，并不是NIH综合征。这种借鉴其他开发者的概念是Linux世界中一种常见做法，当然前提条件是每个人都能得到源代码。事实上，在原有软件的基础上进行加强和扩展也是Unix哲学的核心概念之一。

1.2　Unix的开发

人们为Unix的可移植性感到惊叹，但一开始它并没有这个特性。Thompson最早是使用汇编语言编写Unix的。1972年，他采用了B语言这种具有可移植性的编程语言改写了代码。很明显，他可能是想利用不断涌现的新硬件的优势。1973年另一位AT&T公司贝尔实验室的成员Dennis Ritchie对B语言进行了扩展和调整，将其发展为今天全世界程序员为之爱恨交加的C语言。

同样，Thompson提供了一个后来广为Unix开发人员仿效的范例：背水一战的人们常常能够编写出伟大的程序。在必须编写某应用程序的时候，如果满足条件（1）它必须满足实际的需要；（2）周围并不存在任何了解该如何编写此程序的“专家”；（3）没有足够时间 “完美”完成任务，那么，写出优秀软件作品的概率就比较高。就好比Thompson的情况，他需要使用一种可移植的语言来编写这个操作系统，因为他必须将他的程序从一个硬件架构迁移到另外的架构；他找不到任何所谓的可移植性操作系统专家；他当然也没有足够的时间去把事情做到“完美”。

不过，Ken Thompson一人对Unix哲学整体发展的推动作用毕竟是有限的。虽然他在文件系统结构、管道、I/O子系统和可移植性等领域的设计方面做出了杰出贡献，然而Unix哲学的成型还是众人努力的结果。初期与Unix打交道的每一个人都将各自的专业领域知识应用在Unix上，帮助塑造了这一哲学理念。下面列出了一些贡献者和他们的主要贡献（参见表1-1）。

表1-1 Unix哲学的主要贡献者及其重大贡献

贡 献 者	所作的贡献
Alfred Aho	文本扫描、分析、排序
Eric Allman	电子邮件
Kenneth Arnold	屏幕更新
Stephen Bourne	Bourne shell命令解释器
Lorinda Cherry	交互式计算器
Steven Feldman	计算机辅助软件工程
Stephen Johnson	编译器设计工具
William Joy	文本编辑，类似C的命令语言
Brian Kernighan	正则表达式、编程规范、计算机排版、计算机辅助指令
Michael Lesk	高级文本格式、拨号网络
John Mashey	命令解释器
Robert Morris	桌面计算器
D. A. Nowitz	拨号网络
Joseph Ossanna	文本格式化语言
Dennis Ritchie	C语言
Larry Wall	Patch工具、Perl语言、*rn*网络新闻阅读器
Peter Weinberger	文本检索

上述人士只是Unix这一独特系统中最早和最知名的参与者，实际上最终有数以千计的人参与了Unix的开发工作。几乎每一篇以Unix主要组件为主题的论文，都列举了一大批贡献者。正是这些贡献者共同塑造了Unix哲学，让它逐渐被人们理解接受并流传至今。

1.3　Linux：一个人加上一百万人的智慧

如果说Ken Thompson是Unix的创造者，那么Linus Torvalds就是Linux操作系统的发明人，当时他还是芬兰赫尔辛基大学的一名学生。1991年8月25日他发出了那篇现在广为人知的新闻组主题文章，这篇以“嗨，大家好……我正在编写一个（免费）的操作系统”开头的文章对他的命运产生了深远影响。

Thompson和Torvalds两人至少有一点相似之处，那就是对事物的好奇之心。我们可以找到证据，Thompson编写Space Travel程序只是为了好玩而已。而Torvalds在痴迷于类Unix操作系统——Minix的同时，也完全是因为非常感兴趣才会将流行的Unix命令解释器bash进行改编并运行在他的“玩具”操作系统上。同时，这些在一开始只是“为了好玩”的举动，却最终对软件产业产生了深远影响。

一开始，Linux也不是一款具备可移植性的操作系统。Torvalds无意将它移植到英特尔386之外的其他架构之上。从某种意义上说，他也只是背水一战，因为他的手头只有少量计算机硬件可供选择。因此，最初他并没有采取任何进一步的举措而只是将自己拥有的资源发挥到极致。但是他发现良好的设计原则和扎实的开发模式还是引领着他去把Linux变成一个可移植的系统。从那一刻开始，别的人接过了这个接力棒，很快便将Linux移植到了其他架构。

在Torvalds的Linux出现之前，借鉴他人编写的软件已成为相当普遍的做法。事实上也就是因为这样，Richard M. Stallman才会在具有里程碑意义的GNU公共授权协议（GPL）下正式确立了这一思想。GPL是一个适用于软件的法律协议，基本保证了软件的源代码可以自由提供给任何想要得到它的人。Torvalds最终为Linux采用了GPL协议，这个举动免除了所有人对于相关法律与版权纠纷的后顾之忧，让他们可以自由借用Linux的源代码[①]。由于Torvalds将Linux免费开放出来，因此其他人自然也会将他们的软件免费提供出来以共同发展Linux。

从一开始，Linux已经表现出它确实是一个与与Unix非常相似的操作系统。它的开发人员全盘接受了Unix的哲学原理，然后再从头编写了这个新的操作系统。问

① Torvalds借鉴的东西远不止是GPL。Stallman和其他人都曾经指出Linux实际上只是一个操作系统内核而已。绝大部分处于内核外围的程序都来源于自由软件基金会里的GNU项目。追随者理所当然地将这整套系统叫做“GNU/Linux”，在这套系统中，Linux内核是最为关键的一个组成部分。荣誉应当属于有功人士，我们也意识到大多数人已经作出了自己的选择，将这套系统称为Linux。

题是在Linux的世界里，几乎再没有其他程序是重新编写的。一切应用都是建立在其他人写好的代码和概念之上。因此很自然地，Linux成为了Unix系统演变的下一步，或许更准确地说，它是Unix的一个大飞跃。

类似于Unix，在Linux技术发展的早期，有许多开发者参与其中并提供了帮助。不同的是，Unix开发者数量最多的时候也就几千人，而今天Linux的开发者数量却早已达到了几百万之多。这才是登峰造极的Unix！正是这种大规模的开发格局，保证了Unix的后代Linux将在很长时间内都是一款具备强大竞争力的系统。

Linux为Unix世界重新激起波澜，所谓的“开源”要比“专有”软件或是那些没有现成源代码的软件优越。多年以来，Unix开发人员一直坚信这一点。但计算机行业的其他人却被一些专有软件公司的大量宣传所蒙蔽，他们误认为任何借来的或是免费的软件在性能上都无法比拟那些要付费（有时甚至是耗费巨资）的软件。

在市场营销方面，Linux社区也更为精明，他们知道只要市场工作做得好，就算是劣质软件也可以成功销售出数百万份。当然，这并不是说Linux是伪劣产品。只是，有别于它的前身Unix社区，Linux社区认识到，即使是世界上最好的软件，也只有当人们对它产生了解并认识到它的真正价值时，才会为人所用。

我们将在后续章节再深入探讨这些主题。现在，让我们把Linux和Unix的历史留在过去，继续前行。事情会更加有趣。

1.4 Unix 哲学概述

Unix哲学的几条准则看似简单。事实上，它们简单到会容易使人们忽略其重要性。这就是它们颇具欺骗性的地方。其实，简单的外表下掩盖着一个事实：如果人们能够始终如一地贯彻它们，这些准则可是非常行之有效的。

以下这份清单会让你对Unix哲学的准则有初步的认识。本书其余部分则会帮助你理解它们的重要性。

(1) 小即是美。相对于同类庞然大物，小巧的事物有着无可比拟的巨大优势。其中一点就是它们能够以独特有效的方式结合其他小事物，而且这种方式往往是最初的设计者没能预见到的。

(2) 让每一个程序只做好一件事情。通过集中精力应对单一任务，程序可以减少很多冗余代码，从而避免过高的资源开销、不必要的复杂性和缺乏灵活性。

(3) 尽快建立原型。大多数人都认同“建立原型”（prototyping）是任何项目的一个重要组成部分。在其他方法论中，建立原型只是设计阶段中一个不太重要的组成部分，然而，在Unix环境下它却是达成完美设计的主要工具。

(4) 舍高效率而取可移植性。当Unix作为第一个可移植系统而开创先河时，它曾经掀起过轩然大波。今天，可移植性早被视作现代软件设计中一个理所当然的特性，这更加充分说明这条Unix准则早就在Unix之外的系统中获得了广泛认可。

(5) 使用纯文本文件来存储数据。舍高效率而取可移植性强调了可移植代码的重要性。其实可移植性数据的重要性绝不亚于可移植代码。在关于可移植性的准则中，人们往往容易忽视可移植性数据。

(6) 充分利用软件的杠杆效应。很多程序员对可重用代码模块的重要性只有一些肤浅的认识。代码重用能帮助人们充分利用软件的杠杆效应。一些Unix的开发人员正是遵循这个强大的理念，在相对较短的时间内编写出了大量应用程序。

(7) 使用shell脚本来提高杠杆效应和可移植性。shell脚本在软件设计中可谓是一把双刃剑，它可以加强软件的可重用性和可移植性。无论什么时候，只要有可能，编写shell脚本来替代C语言程序都不失为一个良好的选择。

(8) 避免强制性的用户界面。Unix开发人员非常了解，有一些命令用户界面为什么会被称为是“强制性的”（captive）用户界面。这些命令在运行的时候会阻止用户去运行其他命令，这样用户就会成为这些系统的囚徒。在图形用户界面中，这样的界面被称为“模态”（modal）。

(9) 让每一个程序都成为过滤器。所有软件程序共有的最基本特性就是，它们只修改而从不创造数据。因此，基于软件的过滤器本质，人们就应该把它们编写成执行过滤器任务的程序。

以上列出了Unix开发人员所奉行的信条。在其他一些Unix书籍中你也会找到类似清单，因为这些都是大家公认的Unix基本理念。如果你也采用这些准则，那么人们就会认为你是一个“Unix人”。

下面还列出了10条次要准则，这些准则正在渐渐发展成Unix世界信仰体系的一个组成部分[①]。并非每个与Unix打交道的人都会将它们奉为信条，而且在严格意义上其中一些并不能算作是Unix的特性。不过，它们看起来依然是Unix文化（当然也包括Linux文化）不可或缺的一部分。

(1) 允许用户定制环境。Unix用户喜欢掌控系统环境，并且是整个环境。很多Unix应用程序绝对不会一刀切地使用交互风格，而是将选择的权利留给用户。它的基本思想就是，程序应该只提供解决问题的机制，而不是为解决问题的方法限定标准。让用户去探索属于自己的通往计算机的佳境之路吧。

(2) 尽量使操作系统内核小而轻巧。尽管对新功能的追求永无止境，Unix开发人员还是喜欢让操作系统最核心部分保持最小的规模。当然，他们并不总是能做到这一点，但这是他们的目标。

(3) 使用小写字母，并尽量保持简短。使用小写字母是Unix环境中的传统，尽管这么做的理由已不复存在，但人们还是保留了这个传统。今天，许多Unix用户之所以要使用小写的命令和神秘的名字，不再是因为有其限制条件，而是他们就喜欢这么做。

(4) 保护树木。Unix用户普遍不太赞成使用纸质文档。而是在线存储所有文字档案。此外，使用功能强大的在线工具来处理文件是非常环保的做法。

(5) 沉默是金。在需要提供出错信息的时候，Unix命令是出了名地喜欢保持沉默。虽然很多经验丰富的Unix用户认为这是可取的做法，可其他操作系统的用户却并不认同此观点。

(6) 并行思考。大多数任务都能分解成更小的子任务。这些子任务可以并行运行，因而，在完成一项大任务的时间内，可以完成更多子任务。今天已涌现出大量对称多处理（symmetric multiprocessing，SMP）设计，这说明计算机行业正在朝着并行处理的方向发展。

(7) 各部分之和大于整体。小程序集合而成的大型应用程序比单个的大程序更灵活，也更为实用，本条准则正是源于此想法。两种解决方案可能具备同样的功能，

① 考虑到我给出的这些词语，比如“信条”、“准则”和“信仰体系”，人们或许会思考，Unix哲学除了介绍技术外，是否还描述了一种文化现象。

可集合小程序的方法更具有前瞻性。

(8) 寻找90%的解决方案。百分百地完成任何事情都是很困难的。完成90%的目标会更有效率，并且更节省成本。Unix开发人员总是在寻找能够满足目标用户90%要求的解决方案，剩下的10%则任由其自生自灭。

(9) 更坏就是更好。Unix爱好者认为具有“最小公分母”的系统是最容易存活的系统。比起高品质而昂贵的系统，那些便宜但有效的系统更容易得到普及。于是，PC兼容机的世界从Unix世界借鉴了此想法，并取得巨大成功。这其中的关键字是包容。如果某一事物的包容性强到足以涵盖几乎所有事物，那它就比那些“独家”系统要好很多。

(10) 层次化思考。Unix用户和开发人员都喜欢分层次来组织事物。例如，Unix目录结构是最早将树结构应用于文件系统的架构之一。Unix的层次思考已扩展到其他领域，如网络服务命名器、窗口管理、面向对象开发。

看完这份准则清单，你可能会觉得所有内容都有点儿小题大做。“小即是美”不是什么大不了的道理。“只做好一件事”本身听起来也相当狭隘。“舍高效率而取可移植性”也并不是那种能够改变世界的真知灼见。

这就是Unix所蕴含的道理吗？Linux难道也只是给目光短浅的人准备的小型操作系统？

也许我们应该提及，大众汽车公司曾经围绕着“小即是美”的概念开展了一次成功的汽车营销活动，并借此销售了数以百万计的汽车；或者想一想主流Unix供应商Sun公司的事例，它的商业战略基于“集中所有资源推出最好的拳头产品”这一思想，或者换句话说，也就是“只做好一件事”的理念。那么，人们对于掌上电脑、无线网络访问和手持视频的兴趣是否与可移植性有关呢？

让我们现在就开始这个精彩的旅程吧。

第2章

人类的一小步

2

大约30年前，当美国人边开着大型轿车边享受着其他国家民众的艳羡目光时，大众汽车却在美国开展了一项主题为“小即是美”的广告营销活动。那时，这家德国汽车制造厂商的举动似乎有些不合时宜。要知道虽然大众的甲壳虫汽车在欧洲获得了巨大成功，可是让它们奔驰在美国的广袤大地上看起来可有点儿滑稽，就像是巨人土地上的侏儒。尽管如此，大众汽车却一直不遗余力地向美国家庭传递小型车很适合他们的理念。

然后，令人意想不到的事情发生了：中东的石油部长们向世界表明，是的，他们完全某些事务上完全有自主权——也就是，可以大幅度抬高每桶石油的价格。他们通过大量囤积原油让市场供需的天平倾向供应商的一侧。每加仑[①]汽油的价格猛涨到了1美元以上。全世界那些被中东扼住能源动脉的地区不得已修建了很多天燃气管道。

长期以来，美国人对大轿车的迷恋世人皆知，然而他们也开始意识到，的确“小即是美”。为了缓解日益羞涩的经济状况，他们开始向汽车制造商订购小型汽车，大众汽车公司成了他们的救星。这款昔日里的“可笑小车”变成了时髦的必需品。

随着时间的推移，人们发现小型车的确拥有一些大型车无法比拟的优势。除了能够降低油耗外，小型车的操纵方式也备受大家喜爱——它们就像是英国生产的跑车，而不是长了轮子的远洋客轮。小型车还能轻而易举地挤进狭小的停车位。它结构简单也因此更易于保养维护。

---

① 1加仑≈3.785升。——编者注

就在美国人日益热衷小型汽车的同时，AT&T公司贝尔实验室位于新泽西州的研究小组也发现，小型软件程序也拥有某些优势。他们觉得小程序就如同小型汽车一样，更易于操控，有着更强的适应性，维护起来也比大程序方便得多。这给我们带来了Unix哲学的第一条准则。

2.1　准则 1：小即是美

如果你准备开始编写一个程序，请从小规模开始并尽量保持。无论是设计简单的过滤器、图形软件包还是庞大的数据库，你应该尽自己所能将它的规模降至最小，实用即可。请抵制诱惑避免让它成为庞然大物。软件开发应该力求简单。

传统程序员的心中经常怀有一个编写出伟大程序的隐秘渴望。在着手准备开发项目时，他们仿佛想要花费数周、数月甚至数年的时间去开发一个能够解决世界上全部问题的程序。这种做法不仅商业代价高昂，而且也脱离现实。其实在现实世界中，只要把一些小巧的解决方案组合起来，几乎不存在解决不了的问题。之所以会选择实施如此大规模的解决方案，根本原因在于我们并没有完全理解该问题。

科幻小说作家Theodore Sturgeon曾写道："90%的科幻小说是垃圾。但其实所有事物中的90%都一无是处。"这同样适用于最传统的软件。任何程序的大部分代码实际上都没在执行它所宣称的功能。

你对此感到怀疑？让我们来看一个例子。假设你想编写一个将A文件复制到B文件的程序。以下就是一个典型的文件复制程序可能要执行的步骤。

(1) 要求用户输入源文件的名称。
(2) 检查源文件是否存在。
(3) 如果源文件不存在，提示用户。
(4) 询问用户目标文件的名称。
(5) 检查目标文件是否存在。
(6) 如果目标文件存在的话，询问用户是否需要替换该文件。
(7) 打开源文件。
(8) 如果源文件内容为空，提示用户。必要的话可以退出此程序。

(9) 打开目标文件。

(10) 将源文件的数据复制到目标文件里。

(11) 关闭源文件。

(12) 关闭目标文件。

请注意，文件只在步骤(10)中才真正被复制。其他操作步骤虽然也很有必要，却与复制文件这个目标并无直接关联。如果仔细研究，你会发现这些其他步骤也可以应用于除文件复制之外的众多任务。这里只是恰好也要用，但并非是任务的真正组成部分。

优秀的Unix程序应该只提供类似于步骤(10)的功能，别无其他。进一步来说，一个严格遵循Unix哲学的程序会在调用时就已获取了有效的源文件名和目标文件名。它应该只负责复制数据。显然，如果程序所做的事情只是复制数据，那的确只会是很小的程序。

我们仍然有疑问，那程序中有效的源文件名和目标文件名来自何方呢？答案很简单：从其他小程序那里获取。这些小程序将执行不同的功能：获取文件名，检查文件是否存在，或确定其中内容是否为空。

“等一下！”你可能会想。我们是不是在说Unix里面存在着只检查文件是否存在的程序？没错。标准的Unix发行版本包括几百个小命令和小型应用程序，它们自身只能完成一些简单功能。类似test命令这种执行平常小任务的程序很多，比如只确定文件的可读性，等等。这听起来似乎无足轻重，但要知道test命令可是Unix上使用得最频繁命令之一[①]。

就自身而言，小程序做的事情并不太多。它们经常只实现一两个功能。但要是将它们结合在一起，你就能体会到它们真正的力量。它们的整体功能大于各个局部功能的简单相加。大型复杂的任务就此迎刃而解。你只需要在命令行上输入这些小命令，就可以编写出一个新的应用程序。

① 一些Linux/Unix的shell功能（命令解释器）比如bash就嵌入了test命令，人们不再需要调用新进程来执行这个命令，从而减少了系统资源的开销。不过这样做的缺点就是，如果你一直往shell中添加命令，它的规模会不断扩大，直到运行非shell命令的代价因Linux/Unix所采取的创建新进程方式而变得非常高昂。所以，最好的做法就是将经常使用的命令驻留在内核的高速缓存中，这样就不需要从磁盘中获取这些命令，从而避免巨大的时间开销。

2.2　简化软件工程

人们常常说，Unix给程序员提供了世界上最丰富的系统环境。原因之一就是，那些在其他操作系统上很难执行的任务在Unix上却能轻松实现。是小程序让这些任务变得简单了吗？是的，这毋庸置疑！

2.2.1　小程序易于理解

小程序只会集中精力完成一项功能，因此将小程序集合在一起的“全业务”解决方案能将失误减到最少。它们只包含为数不多的几个算法，绝大部分都与要完成的工作直接相关。

另一方面，大程序相对更复杂，也给人们带来了理解障碍。程序的规模越大，就会越发背离作者的初衷。代码的行数也会多得令人难以忍受。比如，程序员可能会忘记某个子程序到底在哪些文件中，或是难以交叉引用变量并记住它们的用法。调试代码也会成为噩梦。

当然，总会有一些让人难以理解的程序，不管它们的规模是大是小，这是因为本身它们执行的功能就晦涩难懂。但这样的程序并不多见。一般来讲，中等水平的程序员都能毫不费力地理解大多数小程序。因此，相比大程序，这可以算作是小程序的一个明显优势。

现在，你可能想知道小程序什么时候会变成一个大程序。答案就是，视情况而定。此环境下的大程序在彼环境中也许只是个中等规模的程序；而且，同样的代码在这个程序员眼里算是大程序，而在别的程序员眼中可能只是小菜一碟。下面列举了一些迹象，如果你编写的软件有出现这些迹象，就表明该软件已经偏离了Unix的基本操作思路。

- 传递给函数调用的参数数量过多，导致代码超出了屏幕的宽度。
- 子程序代码的长度超过了整个屏幕或是一张A4纸的长度。注意，使用较小的字体且在窗口较大的工作站显示器上，你可以有空间扩展一下代码量。只是请不要得意忘形。
- 要靠阅读代码注释，你才能记住子程序到底在做些什么。
- 在获取目录列表的时候，屏幕显示不下这些源文件的名称。
- 你发现某个文件已经变得很难控制，无法定义程序的全局变量。

- 在开发某程序的过程中，你已经无法记起一个给定的错误信息是在什么条件下引发的。
- 你会发现自己不得不在纸上打印出源代码才能更好地组织思路。

这些警示可能会激怒一些程序员，也就是身处“大就是好”阵营的那一拨人。他们会争辩不是每个程序都能成为小程序。我们的这个世界如此巨大，总有一些需要使用大型机去解决的超大问题。所以，我们需要编写宏大的程序来解决它们。

这是一个很大的误区。

往往有这样一类软件工程师，为自己能编写出规模宏大的程序而深感骄傲，可除了他自己，没有任何人能读懂这些程序。他会认为只有这样才具有“职业保障”。可以这么说，就只剩他编写的应用程序要大过其自负情结。在传统软件工程环境中，这样的软件工程师可以说是屡见不鲜。

通过这种手段来确保饭碗会带来如下问题：这些人效力的公司一定会意识到，他们最终还是会跳槽到别的地方而将烂摊子留给公司。明智的公司知道采取一些措施来防止这种情况的发生。他们会聘请那些真正懂得易于维护的软件才更具价值的人。

软件设计师再也不能说：“上帝保佑接手的那个可怜虫。”优秀的设计师必须不遗余力地追求软件的可维护性。他们会给自己的代码加上完整而不冗余的注释。他们会尽量编写短小精悍的子程序。他们大幅削减代码，只留下绝对必要的那些部分。经过如此努力产生的作品一定是易于维护的小程序。

2.2.2 小程序易于维护

小程序通常更容易理解，也就更易于维护，因为理解程序是软件维护的第一步。毫无疑问，你肯定听说过此观点，但是有许多程序员会忽视软件维护问题。他们认为，既然自己费时费力写好了程序，那么别人也同样会愿意花点儿时间去维护它。

大多数软件工程师不会满足于靠维护他人软件为生。他们认为（也许真是）只有编写新软件才能真正挣到钱，而不是靠修复旧有软件的bug。不幸的是，用户却不这么想。他们希望软件能够稳定运行。如果软件无法做到这点，用户就会对程序供应商产生极大不满。要知道，那些无法好好维护软件的公司可经营不了太久。

软件维护不是一份很吸引人的工作，因此，程序员一直在想办法让这个任务变得更容易，甚至干脆避开它。然而，大部分人都无法推卸这份责任。幸运的话，他们也许能够将这份烦人的工作交给新手来做。但更多时候，软件维护这一职责还是会落在原作者身上，他必须尽其所能让这个任务变得更为轻松愉快。小程序则完美地满足了这一需求。

2.2.3　小程序消耗的系统资源较少

因为它们的可执行镜像只占用了少量内存，操作系统就更能轻而易举地为它们分配空间。这大大降低了内存交换和分页的需求，往往能够显著提高系统性能。“轻量级”是一个在Unix世界颇为流行的术语（比如，我们一般都认为小程序就是一个轻量级的进程）。

大型程序则有着庞大的二进制镜像文件（binary image），操作系统要花费巨大代价来装载它们。频繁发生的分页和交换动作，进而严重影响性能。为了解决大型程序的资源需求，操作系统设计者试图通过创建动态加载库和共享运行库等功能来对此作出改善。不过，这些都只治标不治本。

在职业生涯的初期，我碰到过一位计算机硬件工程师，他经常会开玩笑地说：“所有的程序员都想要更多的核心资源！我们只能再多加一条内存，才能堵住他们的嘴。”其实，他这个玩笑还是有一定道理。给程序员更多内存，程序就会运行得更快一些，编写程序的时间也就更短，这样会大大地提高生产效率。

让我们来反思一下这个“更多核心资源”的解决方案。在说上述那段话时，我的硬件工程师朋友无意中透露了他的判断依据，即他周围的程序员偏爱编写大型、复杂的程序。这并不奇怪。那会儿我们用来开发应用程序的操作系统平台并不是Unix。

如果我们一直持续不断地使用Unix，并全面接受这种小程序理念，就不会提出更多内存需求。那这个笑话就应该是这样的：“所有的程序员都想要更多的MIPS！”（MIPS的意思是每分钟执行上百万条指令，这种衡量CPU性能的方法很流行，却不一定准确。）

为什么在今天的计算机世界里，MIPS度量法会成为一个热点呢？因为Unix应用得越来越普遍，小程序的使用激增。尽管小程序在执行时只占用系统小部分内存，这些额外的CPU马力却给它们带来了最大化的利益。人们只需要将它们加载到内存

中，小程序就能迅速地完成其工作，然后释放掉它们占用的内存供其他小程序去使用。显然，如果CPU的能力不足，那么每个程序就必须在内存中驻留得久一些，然后才能加载别的小程序来完成其他工作。

内存越大，使用小程序的系统就更能从中受益。大容量内存使得更多小程序可以在内核高速缓存中驻留更长的时间，以减少对二级存储的依赖。小程序也更容易放置在高速缓存中。正如我们将在后面看到的，同时运行的小程序数量越多，整体系统的性能就越高。这也构成了Unix哲学的另外一个基本原理，我们将在以后讨论。

2.2.4 小程序容易与其他工具相结合

任何与复杂大型程序打过交道的人都知道这一点。庞大的单一程序仿佛容纳着整个世界。这些大型程序试图替代一个个单一程序来提供人们需要的每一个功能，而这恰恰有碍自己去配合其他的应用程序一起工作。

人们也许会有疑问，能够提供许多种数据格式转换功能的大应用程序难道不比那种只能转换一种数据格式的小程序更有价值？表面上，这听起来像是一个合理的假设。只要这个应用程序恰恰支持你所需要的转换格式，你就会感觉良好。可是，在需要让该应用程序转换某种它并不支持的数据格式时，那该怎么办？

编写大型程序的软件开发人员通常都有一种错觉，误以为他们能够处理各种意外事件（即他们的程序能处理今天存在的任何数据格式）。这是个问题。虽然开发人员能够处理当前的数据格式，但他们并不了解未来会出现什么使他们的程序变得过时的新格式。那些骄傲自大的、编写大型复杂程序的人们不但喜欢预测未来，并且还认为未来和现在不会有什么大不同。

另一方面，那些编写小程序的人则会保持低调，尽量避免去预测未来的状况。他们对明天作出的唯一假设就是它必将和今天不同。未来，新的接口会出现，数据格式也将发展。程序的互动方式也随着大家口味的变化而不同。新硬件技术出来，旧算法就会显得过时。变化不可避免。

总之，与编写大型程序相比，你会发现自己更乐于编写小程序。其简单性也使得它们的编写、理解和维护更加容易。此外，人与计算机都会发现它们更具有包容性。当然最重要的就是，在编写这些程序的同时，你就可以想出方案以应对那些无法预见的情况。

向前看。未来可能会来得比你想象的快。

2.3　关于“昆虫”的研究

让我们花一点时间来探讨一下昆虫（bug）吧——不是软件bug，而是现实世界里的昆虫。

在尼罗河源头维多利亚湖的周围，栖息着一种特别的飞虫。已故探险家Jacques Cousteau曾经拍摄到令人叹为观止的录像，画面中这种飞虫聚集在湖泊上方和附近的丛林里，像一片黑压压的浓雾。这种昆虫大小和外观很像蚊子，有时候它们密集在湖泊上，人们会误以为那是水上龙卷风或是小型飓风。

鸟类经常成群结队地突袭这些“昆虫群”，以享受这大自然恩赐的饕餮大餐。鸟儿们单次空袭所吞噬的昆虫数量高达几百万只。尽管遭遇到这种大规模的捕食攻击，可还是会有很多昆虫得以幸存，从其在空中形成的巨大阴影来看，它们几乎没有消减的迹象。

Cousteau对这些特别昆虫的生命周期做了深入密切的观察。他发现，昆虫的成年阶段极其短暂——大约只有6~12个小时。即使它们能够幸免于难，没有成为那些鸟儿的午餐，其生命也很短暂，只不过能在阳光下振翅的几个小时而已。

那么，这些成年生活只有短短一天时间的昆虫在做些什么呢？繁衍后代。它将这件我们人类需要花上数年的事情压缩在几个小时内完成。很显然它的做法很成功，因为这个物种并没有灭绝，其数量之多令人肃然起敬。如果说种群的生存延续是它们的目标，那么这些长着翅膀的微不足道的小生命的确很了解自己的优先任务。

这些飞虫的一生只做一件事情，而且它们完成得很好。Unix的开发人员认为，软件也应该这样做。

2.4　准则 2：让每一个程序只做好一件事

最好的程序应该像Cousteau所拍摄的湖泊飞虫一样，全部能量只用来执行单一任务，并且将它完成得很好。程序被加载到内存中，行使完它的功能，然后退出，让下一个目标单一的程序开始运行。这听起来很简单，可你会发现，太多软件开发人员无法坚持朝着这个简单方向努力。

软件工程师经常会不由自主地成为“功能控”（creeping featurism），这是业内惯常的说法。一开始，程序员也许只想编写一个简单的应用程序，可随后他的创新精神会慢慢占据上风，促使他在这里添加一个功能，在那里加入一个选项。很快他的程序就成为一个不折不扣的大杂烩，对原有规划而言，大多数功能也无法创造更多价值。也许这些发明中会有一点儿可取之处。（当然我们并不是要扼杀人们的创造力。）但是，开发者必须考虑这些功能是否归属于这片代码。下面一组问题可以为你的决断提供良好依据。

- 这个程序需要与用户互动吗？用户是否要在文件或是命令行中给出必要的参数？
- 这个程序需要输入特殊格式的数据吗？系统上是否有能提供该格式转换功能的程序？
- 这个程序需要输出特殊格式的数据吗？明文的ASCII文本是否就足够？
- 有没有其他程序可以执行你想要完成的类似功能，而不需要重新编写代码？

前3个问题的答案通常都是否定的。真正需要直接与用户交互的应用程序比较罕见。大多数程序都能良好运行，完全不需要在它们的常规活动中加入对话解析器。同样，只要使用标准的输入和输出数据格式，大多数程序就可以满足大多数需要。在那些需要特殊文件格式的情况下，我们完全可以使用其他通用程序来作数据转换。否则，我们需要为每一个新程序另起炉灶。

Unix的`ls`命令是绝佳范例，可以充分说明Unix应用程序是怎样误入歧途的。截至到目前，它已经拥有了二十多个选项，而且一直在增加。伴随着每个新版Unix的出现，它都会有更多选项。我们暂且不理会那些稀奇古怪的功能，先来看看一项更基本的功能，也就是它格式化输出结果的方式。最纯粹的`ls`命令应该这样列出一个目录中的文件名（未经排序）：

```
/home/gancarz -> ls
txt
doc
scripts
bin
calendar
X11
database
people
```

mail
slides
projects
personal
bitmaps
src
memos

然而，大多数版本的ls命令都是按照如下格式输出结果：

mail	calendar	people	slides
X11	database	personal	src
bin	doc	projects	txt
bitmaps	memos	scripts	

一开始，将文件名显示在整齐的列中似乎是种明智做法。但现在ls的代码里面还包括了列格式化的功能，这个任务与列出文件目录内容已经没有太大关系。列的格式化可以很简单，也可以很复杂，取决于它的使用环境。例如，默认状态下，ls假定用户正在使用80个字符宽的旧式字符终端。那如果在一个132字符宽的终端屏幕的窗口系统上调用ls命令，会是什么样子？假设用户喜欢将输出结果分两列显示，而不是四列？假设在终端使用可变宽的字符集呢？假设用户希望在每显示完五行文件名之后有一条实线作为分割呢？这样的情况不胜枚举。

平心而论，ls命令还是保留了逐行显示文件夹下各个文件名的能力。其实这就是它应该做的全部工作，我们完全可以把列格式化的工作交给其他的命令。这样，ls命令就会小得多（也就会更易于理解、易于维护、占用较少的系统资源等）。

编写这种“只做好一件事”的应用最终会产生一个较小的程序，因此这两项原则是相辅相成的。小程序往往只具有单一功能，而单一功能的程序往往也会很小。

这种做法的潜在好处就是，你可以集中精力去解决当前的任务，全心全意做好本职工作。如果无法让程序只做好一件事，那么你很可能并不理解自己正在试图解决的问题。接下来的一章中，我们将讨论如何以Unix的方式去理解问题。现在，让我们着眼于小处，只做好一件事。

如果尼罗河上的湖泊飞虫都能做到这一点，那对我们来说又会有多难呢？

第3章 快速建立原型的乐趣和好处

拉玛人喜欢三五成群地去做一切事情。

——选自阿瑟·克拉克的*Rendezvous with Rama*[①]

3.1 知识与学习曲线

漫步在华尔街，你很快就会发现那些“菜鸟”投资者根本不知道自己在做些什么。要在股市中挣到钱，大家都知道必须“低买高卖”。然而，年复一年，“豺狼们”还是从可怜的“羊羔们”那里搜刮着数以百万计的美元。小羊羔们（指的就是你和我）经常在市场关键的转折点踏错节拍，这些可都是有据可查的。

其实机构投资者的表现也不见得就好多少。大多数养老基金经理、共同基金投资组合经理和专业理财人士在市场中的表现同样是一年不如一年，尽管其中很多人的年薪都已超过了百万美元。

研究表明，指数基金的长期表现要比77%的共同基金好，指数基金是指以某一指数的成分股为投资对象的基金，如标准普尔500指数（Standard & Poor's 500）。尽管世界金融市场在大肆炒作投资热点机会，已公布的记录却阐明了一个严峻的现实：哪怕我们在报纸上的股票版面随机购买一些股票，大多数人的表现也会可圈可点，不逊于任何其他投资者，不管他们是业余股民还是专业人士。

虽然在股票市场一直立于不败之地很难，可有些人还是做到了。比如，富达麦哲伦基金（Fidelity Magellan Fund）著名的前任经理彼得·林奇（Peter Lynch），他

① 该书中文版名为《与拉玛相会》，已由四川少年儿童出版社于1998年出版。——编者注

在20世纪80年代累积创造了令人惊讶的交易记录，让他的客户都变得非常富有；“股神”沃伦·巴菲特（Warren Buffett）有着“奥马哈先知”的美誉，也为伯克希尔·哈撒韦公司（Berkshire Hathaway）的股东获取了巨额利润；约翰·邓普顿爵士（Sir John Templeton）则在世界各地寻求投资的绝佳机会，同样谋得了巨大财富。

尽管他们都很成功，可这些传奇式的投资者也坦言，他们并不总是无往不利。他们会讲述自己失败的投资经历，在他们购买了那些自己感到信心十足的公司的股票后，一年内这些股票的市值却狂跌一半。他们会发出感慨：“该买的没买！”而那些不太有希望的股票反倒暴涨了10倍或是更高。虽然他们的整体表现仍然远超普通投资者，但他们知道，自己还是有很多东西要学习。

其他专业人士也需要掌握很多新知识。医生必须努力跟上医学研究方面的最新发展。会计人员需要学习税法的新变化。律师得研究新的法院判决案例。精算师、销售人员、卡车司机、装配工、水管工、时装设计师、法官、研究人员、建筑工人和工程师都需要不断地去学习新知识。

3.1.1　事实上，每个人有自己的学习曲线

想一想吧。你最近一次碰到一个能准确无误了解每次行动结果的天才是在什么时候？我并不是说天赋异禀的人不存在。我只是想表明，天才是非常罕见的。杰出的才能通常需要付出艰苦的努力和学习，再加上一点点好运气。

工程师可能是最佳范例，用来证实大多数人还在学习这一事实。例如，如果航空工程师了解航空领域的所有知识，那为什么他们还是需要试飞员？为什么通用汽车公司的工程师需要对他们的车进行“道路测试”？为什么电脑工程师必须要把他们的产品带到现场作测试，然后才投入大规模的生产？如果工程师们对他们的工作有十分把握，那就不需要质量保证部门，因为开发阶段就应该能保证产品的高质量。

如果这样的话，Unix工程师们就可以使用cat[①]命令来编写程序了。

软件工程师尤其需要高强度持续不断地学习，他们有着一条“陡峭”的学习曲线。软件很难一蹴而就。软件工程这项工作包含着不断的修订过程，在这个行当中，

① 所有Unix命令中最简单的一个，cat命令会把用户键入的所有内容都导入到一个文件。有别于文本编辑器或文字处理器，cat命令不能修改已经输入的文本。

试验和错误都屡见不鲜，应用程序的最终成型来之不易，它建立在令人沮丧的无数个小时的返工之上。

请注意，我们并不是说人们永远无法真正地精通某个行当类。只是，它所需要的时间比大多数人想象的要久得多。人类的平均学习曲线进一步延伸下去会比第一次出现的时候更加“陡峭”。当今世界存在着如此多的变数，想真正掌握某些知识有时候需要穷极人的一生，而且人们不可能学会所有的事物。

3.1.2 大师们也知道，变化不可避免

几乎没有什么项目会一直遵守原有的规格说明书，而不发生任何变化。营销需求会改变，供应商不能提供产品，关键部件的表现可能会与原设想完全不同。因此，建立原型并做测试能够暴露设计上的缺陷。正是这些因素使得复杂技术产品的开发工作成为一项最需要小心处理的任务。

发生变化是不可避免的，这通常要归咎于人们沟通交流的失败。当产品的最终用户试图解释其需求时，他的叙述与实际情况会有出入。他可能忽略了一些事情，或是没能精确表达出他脑海里那些想法的细节。因此，工程师只能用想象力来填补这些空白。尽管他会仔细研读需求文档，但是在设计过程中难免还是会带上自己的主观偏见。有时候工程师试图去猜测最终用户的想法或假定“他是想这样做，而不是那样做”。更糟糕的是，工程师与最终用户之间往往还隔着一些人，比如产品经理、销售团队、支持人员，这些人会进一步曲解最终用户的期望。

人的知识总是有限的。除了雾里看花之外，现实经验也决定了我们不会随随便便成功。大家应该坦然面对这个现实。在设计过程中，我们就要考虑到未来必将发生变化。这样，对目标了解得越充分，就越能降低对产品做重大修改的成本。

3.1.3 为什么软件会被称为“软件”

软件工程比任何其他工程学科都更需要返工，因为软件涉及到抽象概念。如果准确描述硬件都会有困难的话，那可以想象一下人们形容那些只存在于脑海里的想法或芯片中电流的传导模式该是多么困难。我一下就想到了一句格言：“入此门者，莫存希望”（Abandon all hope, all ye who enter here）。

如果最终用户可以详细阐明其想要的功能，如果软件工程师能够完全了解用户

现在和未来的需求，那我们可能就不需要软件了。每个编写出来的程序可以第一时间就被烧录到只读存储器（ROM）里。遗憾的是，这种完美的世界并不存在。

在早期做软件设计师的日子里，我常常会追求软件的完美。我反复修改子程序，直到它们能够最快速地运行，代码也足够干净整洁；我还重复审阅程序，希望能找出哪怕一丁点的改进之处；在想到任何新点子的时候，我都会去添加新的功能（是的，我就是传说中的“功能控”）。我尽情发挥，直到老板将我拽回到现实。

“该发布软件了。”他宣布。

“可是我还没有干完呢！再多给我几天时间吧……”

“永远都没有做完的软件，只有发布的软件。”

在软件发布之后，没有人真正知道会发生什么。经验丰富的工程师可能会有一些模糊概念，大概知道软件面对的客户群，它会被应用于什么样的环境之下，等等。但他们很难预料这款软件产品最终的命运到底是成功还是失败。

我一度在一家大型电脑公司担任电话技术支持工程师。在干过多年操作系统的设计工作之后，我转换成了一个与以往截然不同的角色，开始为我们编写的程序作客户支持。这段工作经历确实让我眼界大开。从事软件设计工作时，你其实很容易掉入一个怪圈，认为一切尽在自己的掌握，误以为自己很了解人们会如何使用你的软件。

你错了。也许你确实对此信心十足。但是，安然待在办公室工作间的工程师，身边围绕的同事与自己一样同样拥有高学历，他能意识得到下列诸多情况吗？

- 某个在交易日导致系统反复崩溃的内核bug可能会给一家华尔街企业造成每小时100多万美元经济损失。一个星期之后，该公司在心烦意乱的情况下选择安装原系统竞争对手的产品。可再过了一个月，他们还是重新切换成最早的系统，因为尽管它有一些缺陷，但在这两个系统之间，还是原来的系统显得更为可靠。
- 一家石油公司可能会使用某个系统来对墨西哥湾进行地震波扫描以寻找天然的石油储备。他们每天都会获取到千兆字节数据，这需要在系统集群上安装90多块大硬盘。人们必须能够随时获取所有数据。而且出于安全考量，没

有任何一名地质学家能够持有访问所有关键数据的密钥，这是因为公司很担心会有人带着在哪开采石油的机密信息跑掉，这将会给公司造成数十亿美元的损失。

- 一家每天在数据库中存储超过1000万条记录的电信公司想设计一个新系统，让它每天可以插入1亿条记录。每秒处理1000次数据库的插入动作本身就已经够困难的了，可现在，他们不得不在高峰期处理每秒高达5000次的数据插入工作。
- 某家电力公司的系统是美国最大一个州的电网中央控制器。一旦该系统出现故障，就可能导致百万人断电。

Unix的开发人员当然不知道Unix上会发生什么事情。MS-DOS、OpenVMS和一切其他操作系统的开发人员也一样。每一个新的操作系统（Linux也不例外）都会将它们的设计者带到未知领域。他们唯一的希望就是不断收集系统使用报告，然后纠正过程中的相应偏差。

例如，Ken Thompson在编写第一个Unix内核时，有没有注意到可移植性的重要性呢？显然没有，因为他使用的是汇编语言。后来，他采取了一门高级语言对它进行了改写，也改变了原来的方向。那么，Dennis Ritchie是否预料到了C语言将成为一门通用编程语言，让数百万的程序员为之爱恨交加呢？也没有。他那本经典著作*The C Programming Language*的修订版包括了对该语言规范的修订，这更加让我们确信他在第一次创作的时候并没有将它完全折腾好。

因此，我们大多数人都还在学习。即使我们自负到认为自己知晓一切，也总会有人改变对我们的期望。那我们到底该如何开发软件呢？接下来的这条原则是关键。

3.2 准则 3：尽快建立原型

“尽快”就是越快越好，火速进行。你可以先花少量时间规划这个应用程序，然后便可以创建原型。开始编写代码吧，就好像你的生命完全取决于这个原型一样。记住要趁热打铁，我们根本就没有时间来浪费！

这个想法与许多人认为的“适当工程方法”（proper engineering methodology）背道而驰。大部分人都曾被要求，在进入编程阶段之前应该充分考虑好自己的设计。

他们说："你必须撰写一份功能规格书，并定期对此进行复核，以确保自己没有脱离正确的轨道。撰写一份设计规范可以理清自己的思路。在你开启编译器之前，应该确保90%的设计已经完成。"

大体上，这些原则听起来都不错，但它们都没有给予原型足够的重视。只有建立原型，你的构想才能首先通过可视化、现实可行的方法得到验证。在此之前，它们只不过是你脑海中一些零零碎碎的想法。那时候，你对这些概念几乎没什么认识，同时，别人也不可能像你一样去理解问题。你需要大家在项目进行之前达成共识。通过原型来提供一个关于目标的具体化表象，会得到客户认同。

3.2.1　原型的建立是学习的过程

越早开始建立原型，就离可发布产品的状态越近。原型可以显示哪些想法可行，最重要的是，哪些是不可行的。你需要这种对于你所选开发思路的肯定或否定。在早期发现设想有误，只会令你遭受小小的挫折，这要远远好于到后期才发现问题。比如，大家花了好几个月召开产品开发会议，却完全没发现潜伏在其中的巨大设计缺陷，而等到产品发布日期前3个星期问题才暴露出来时，会让你措手不及的。

3.2.2　建立早期原型能够降低风险

有了一个具体原型，你就可以指着它说道："产品将会是这个样子。"如果能把它演示给值得信赖的用户看看，并获得他们的反馈，你就会了解你的设计是否具有针对性了。很多时候，你会收到大量批评，这没什么关系，因为你已经获得了宝贵的反馈信息。让一小群人批评你总好过在产品发布后听到万千用户要求召回软件而汇聚在一起的愤怒呼吁。

每一个正确设计的背后都有着数百个错误的设计方案。我们可以在早期剔除掉不良设计方案，形成了优胜劣汰的筛选过程，该过程会引导你越来越接近有质量保证的成品。你可以尽快发觉那些无用的算法、总是丢一个节拍的计时器，以及无法与用户进行交互的界面。这些试验可以帮助你"吹尽狂沙始到金"。

大多数人都同意建立原型有很多好处。即使是那些教授传统软件工程方法的学者们也都很快认识到了它的价值。然而，原型只是用来完成目标的一个手段，而非目标本身。建立原型的主旨应该是创建一个我们称之为"第三个系统"的产品。不

过，在讨论该“第三个系统”之前，我们需要谈谈在它之前的两个系统。

3.3 人类创造的三个系统

人类只具备创建三个系统的能力。不管如何努力，无论为之奋斗的时间有多久，人类最终都会意识到想要打破这个规律只是徒劳。人类根本无法建立第四个系统。只有自欺欺人的家伙才不相信这个铁律。

为什么只有三个？这是个难以回答的问题。人们或许能从科学、哲学和宗教的观点出发，得到一些猜测性的理论。每个理论都会对这种情况给出一个貌似合理的解释。但是，最简单的解释可能就是，人类的系统设计过程就像人类自身一样，都必须要经历三个生命阶段：未成年、成年和老年。

在未成年阶段，人们普遍充满活力。他们就像是街区里新来的孩子：浑身活力四射，渴望受到关注，而且显示出无穷潜力。在一个人从未成年阶段发展到成熟阶段的同时，他会成长为对这个世界更加有用的人。他们的职业生涯逐渐成形，与他人的长期合作关系得到发展，在世俗事务中的影响力越来越大。这个人开始给人留下深刻印象——好的坏的或是其他。待到年老的时候，这个人会丧失掉年少时的许多机理能力。而且，随着体格机能的逐渐衰退，其对世俗的影响也会消退，职业生涯都将成为往昔记忆。人们开始抗拒这种改变。然后，沉淀下来的就是那些基于人生经历的宝贵智慧。

人类设计的系统也经历着同样的发展阶段。每个系统都具备与人类生命发展阶段所对应的那些特性。所有的系统都遵循着这条发展路径：从未成年开始，过渡到成熟阶段，直到以老年作为终结。

正如一些人无法活到天年一样，总有一些系统到达不了成熟期。通常，这是由外部环境所决定的。开发计划也许会改变；项目经费可能会被撤销；潜在客户或改变主意，决定购买别家供应商的产品。任何这些因素都有可能导致系统中途夭折。然而，在正常情况下，人们还是可以引领这些系统经历所有的三个发展阶段。为了阐明这个道理，我们将这些阶段称为人类的三个系统。

大多数Unix开发人员并没有听说过人类的三个系统这个说法，但他们的做法却表明这三个系统的确是无处不在的。让我们仔细研究一下它们的一些特性吧。

3.4　人类的“第一个系统”

3.4.1　在背水一战的情况下，人类创建了“第一个系统”

通常情况下开发人员担负着巨大压力，他们必须要赶在截止日期前交付项目，或是满足其他一些时间紧迫的需求。种种压力会激发出他内心灵感创意的火花。最终，在他深思熟虑良久，并在脑海中反复琢磨他的构想之后，这一点火花变成了一小团火焰。工作仍在继续。他的创意本能开始占据上风。灵感的火焰变得越发明亮。

某一刻，他会意识到一部分构想并不只局限在“达成目标”这一范围内。他感觉自己好像在反复思量一些更重要的事物。此时，如果他认为这些构想可以提供更合理的解决方案，那么他就会逐渐淡忘最早制定的目标。

3.4.2　没有足够的时间将事情做好

如果有充足的时间，那他就不会受迫于赶进步的压力。在有压力的情况下就只能靠临场发挥了。临场发挥虽然是一种妥协之举，可他却只能毫无妥协地勇往直前——哪怕朝着错误的方向。至少，他的旁观者都是这样认为的。当开发人员时间不够，只能背水一战的时候，他就可能会打破常规去行事。当然，在他那些思想保守的同事看来，他已经完全丧失了理智。

通常他会受到人们的指责。“他不能这样侥幸行事!”他们坚称，“他根本不知道自己在做什么。他错得太离谱了。”那他又是如何回应的呢？“是的，它很丑，但它管用啊!”

时间的匮乏迫使他必须集中精力去处理这个任务中的重要事项，并忽视掉其他细枝末节。因此，他计划将一些细节留给后续版本去实现。可是我们要注意到，他可能永远不会有机会去完成所谓的“后续版本”。但是，正是因为相信自己在未来会“填补空白”，他才不会脱离正确的轨道，而这个理由也变成了人们为第一版所有缺点作辩护的藉口。

3.4.3　“第一个系统”是单枪匹马或是一小群人开发的

这其中一个原因要归咎于主流世界里许多人对“第一个系统”开发人员所做的事情不屑一顾。这些人无法立足于他所在的高度去看待事物，所以也就理解不了他

的兴奋之情。因此，他们得出结论，他做的工作很有趣，但并未有趣到足以让他们自己也投身其中。

为什么人们会避免参与“第一个系统”的开发工作？第二个原因更为实际：建立“第一个系统”有很大的风险。没有人知道“第一个系统”是否具备能够过渡到“第二个系统”的特性。失败的可能性总是超过80%。用业界行话来说，与失败的“第一个系统”联系在一起可是颇有职业风险的。因此，一些人宁可等到系统构想被证实可行之后才会开始行动。（后面我将进一步讨论，他们通常会成为“第二个系统”的开发人员。）

随之而来的风险就是，不了解真相的经理很有可能会宣称这“第一个系统”可以作为正式产品，并过早地将它交付给市场部门。这通常会导致销售人员过分炒作这个还不是很完善的系统，而且它的缺陷也会暴露无疑。系统会频繁崩溃，往往还发生在最尴尬的时刻。因此，用户会对这个系统产生强烈偏见，认为它是一款质量很差的产品。最终，这还会严重影响到系统的销售状况。

当一个小组建立“第一个系统”的时候，他们热情高昂，很快就能把事情办好。精诚协作的力量注入到这个团队，最终促成了一个强大有凝聚力的整体。团队成员们拥有共同的信念，心怀热切渴望来促使系统成型。他们了解自己的目标并为之孜孜不倦地工作。在这样一个亢奋的环境里，人们感到精神振奋的同时也会精疲力尽。一旦系统成功，它还能会给人们带来巨大的成就感。

有一点是肯定的：“第一个系统”几乎不太可能是由一大群人完成的。一旦团队规模大到影响到成员之间的日常互动交流时，工作效率就会降低，人与人之间也会产生冲突。成员们开始有着各自隐秘的日程安排。而且，在大家开始追求一己私利的同时，就会形成个人的“一亩三分地”。诸如此类事件总是会淡化系统最终目标，使其难以实现。

3.4.4 “第一个系统”是一个“精简、其貌不扬的计算机器”

它用最小的成本获得差强人意的性能。它的开发人员在大多数时候不得不采取权宜之计，被迫通过硬编码（hard-wire）方式来编写应用程序的很多代码，以牺牲功能和灵活性为代价来换取简洁和速度。华而不实的修饰性功能则都留给下一个版本。任何与系统目标无关的事物都被排除在外。软件确实能完成它的使命——也就仅此而已。

当人们将常见的成熟系统和这“第一个系统”作比较的时候，他们通常会对后者的高性能啧啧称奇。他们很想知道为什么自己的系统比不上这个“新来的孩子”。用流行标准来测试的话，这个不起眼的新产品在性能上超过了他们最喜爱的产品，人们对此有些扼腕。这似乎有点儿不公平，事实上也是如此。将“第一个系统”与那些已经第*n*次发布的系统放在一起比较，其实有点儿“关公战秦琼”，因为各自的设计者有着不同的目标。

“第一个系统”的设计者们高度集中精力以解决目前的问题，让某些东西能工作，甚至也不管是什么功能先运行起来再说。后续的设计者通常要花费大量时间来添加新功能，以满足他们察觉到的市场新动向。就整体而言，这些功能往往会对系统性能产生负面影响。是的，后续系统能完成的任务会更多，但是人们需要为这些新功能付出一定的代价。

3.4.5　“第一个系统”的概念可以激发他人的创造力

“第一个系统”使人们陷入了天马行空的思考，“如果……就会怎么样？”它让人们的胃口大开：想要更多的特性、更多的功能、更多的一切。人们会发表一些冒进言论，“想一想它的可能性！”或“想象一下，在子虚乌有生物科技公司我们能用这个系统来完成的事情！”

下面列出了一些领域和技术，在这些领域中人们充满了创意非凡的想象。眼下，它们已经派生出了许多“第一个系统”。

- 人工智能
- 生物技术
- 数字成像
- 数字音乐
- 电子货币系统和无钞社会
- 基因工程和克隆技术
- 因特网和万维网
- 交互式电视
- 火星登陆计划
- 微型机械
- 纳米技术

- 质量管理（六西格玛、全面质量管理等）
- 虚拟现实
- 无线技术

这些能够激发他人想象力的概念成为了“第二个系统”紧随“第一个系统”产生的主要原因。如果“第一个系统”并没有什么令人兴奋的东西，那人们得出的结论都会是：它的确符合某些人的需要，但是这些人（打着哈欠）表示还有更合适的工具。很多“第一个系统”早已夭折，因为它未能激励其围观者采用它的概念来实现其他伟大而美好的事物。

SourceForge（http://www.sourceforge.net）一个流行的网络资源库，展示的都是一些新型“第一个系统”。其中一些软件曾让人们为它们显示出来的新概念或新技术而感到欢欣鼓舞，并围绕其开展了大量相关活动。遗憾的是，不是所有展现出来的事物都符合“第一个系统”的成功标准，其中一些根本无法带给人们任何启发。因此，SourceForge也成为了各种各样“第一个系统”的网络墓地。

3.5 人类的“第二个系统”

“第二个系统”是一个怪胎。在人类创建的三个系统中，它最受关注，而且往往会取得商业上的成功。对于那些喜欢规避风险的个人来说，它的确提供了某种程度上的安全感，而且让人们更容易上手使用。取决于市场的规模，它可能获得数千甚至数百万用户的青睐。然而，颇具讽刺意味的是，在许多方面“第二个系统”其实是三个系统中最差的那个。

3.5.1 “专家”使用“第一个系统”验证过的想法来创建“第二个系统”

“第一个系统”早期的成功深深地吸引了一些人，他们积极参与进来并希望将自己的名字与这个系统紧密相连，从而获得这样那样的回报。每个人都想和成功产品搭上关系。

这个自封的“专家”群体通常包括许多对“第一个系统”颇有微词的批评家。这些人为自己没能参与“第一个系统”的设计工作而深感懊丧，他们发泄着对“第一个系统”创始人的不满，并声称自己完全可以做得更好。有时候他们是对的。他们确实可能在系统设计某些特定方面做得更好。在重新设计“第一个系统”中几个基本算法的时候，他们的专业知识派上了用场。但请记住：“第一个系统”的设计

者（们）并没有足够时间去把事情做好，而这些专家中的许多人一来知道什么是正确的做法，二来也有充足时间和资源来做好它。

另一方面，他们这些“吃不到葡萄”的专家会对别人来之不易的成就大倒酸水。这里面大有一些NIH的意味，也就是曾经流行的“非我发明”综合征。虽然其中许多人其实也有能力去建立“第一个系统”，他们只是没能抢得先机而已。他们参与“第二个系统”的开发工作不是为了找乐子，而是希望借此机会用自己的设计机制来取代“第一个系统”的机制，从而“改善原有设计师明显的业余尝试”，为自己获得专业口碑。

这种态度往往会引起“第一个系统”设计者的愤怒。偶尔，他们也会奋力反击。流行X Window Sytem系统的先驱Bob Scheifler就曾不客气地回应过认为他的早期设计风格过于随意的批评：“如果不喜欢，你完全可以随意编写你自认为具备行业标准的窗口系统。”

3.5.2　“第二个系统”是由委员会设计的

“第一个系统”是小团队的心血结晶，这个团队的规模通常少于7人。然而，为“第二个系统”的设计工作做出贡献的人却有几十、数百，具体到Linux更是数以千计。“第一个系统”的成功就像一块磁石，它吸引到很多人，哪怕他们只是对这个发人深省的想法有一丁点兴趣。其中确有一些人是真心希望能进一步拓展早期的思想，但大部分人却只是凑凑热闹而已。

“第二个系统”的设计委员会公开开展其业务。它把会议通告发布在大家都看得到的网络信息库和用户新闻组上，他们还会通过其他知名的信息渠道来公布消息。它会发布设计文档，骄傲地展示所有贡献者的名字。该委员还会试图确保所有参与者都能获得应有的荣誉——有时甚至并不是他们应得的。打个比方，如果委员会是在建立一条人行道，那其成员都会希望在水泥地上镌刻上他们的大名。

尽管该团队（而不是因为）举办了这些无关紧要的活动，一些真正的设计工作还是在进行。委员会的一些成员主动参与了主要设计作品的工作，并能够交付高质量的软件；有些人则作为某些软件关键领域的“看门人”，以确保所有的修改不仅能达成预想中的目标意图，而且经过了精心设计；另外一些人则扮演着“魔鬼代言人”的角色，促使委员会提供真实的解决方案；还有一些人表示，他们愿意提供一些有趣的议题供大家讨论，由此帮大家理清新的设计思路。

不幸的是，这种由委员会来负责设计工作的做法有一些缺陷。对于小组（一种至少包括两个人的组织机构）而言，就所有要点达成一致意见几乎是不可能的。为了体现自己的价值，每个参与者都想确保自己对总体设计贡献出一些想法，而不管自己是否具备该设计领域的专业知识。其实这些人的对错并不重要。他们只是出于一己之私，希望能有机会对自己说：“与这些专家在一起，我还是能坚持自己的立场，因此，这让我也成为了一名专家。”人们要的只是一头“敏捷羚羊”，可当你把所有人贡献出的想法累加在一起时，却得到了一个“笨重的大象”。

3.5.3 “第二个系统”臃肿而缓慢

它拥有的特性正好与“第一个系统”相反（即，“第一个系统”是精简的，而这“第二个系统”却像一个步履维艰的巨人）。如果“第一个系统”至少需要1MB内存的话，那么“第二个系统”完全无法运行在任何少于4MB内存的机器。哪怕是运行在1兆赫兹主频机器上的“第一个系统”，其高吞吐量也让人们为之交口称赞，而“第二个系统”的用户却在哀叹，就算机器拥有千兆赫兹主频，它的性能却慢如蜗牛。

“这是因为‘第二个系统’具有更多的功能，”委员会如此辩解，“你的付出是有回报的。”

“第二个系统”的确有着更多特性。它拥有一系列令人印象深刻的功能，这也是它赖以成功的原因之一，但普通用户用得上的功能只是其中的一小部分，其他的反而碍事。“第二个系统”运行得非常缓慢，因为它必须耗费资源来处理大量冗余的“优势”。

通常，我们只能通过购买更多硬件来让它运行得更快一些。我一直相信，计算机制造商对“第二个系统”的热爱正是出于此原因。在可移植软件占据主流的今天，大多数“第二个系统”都能运行在几乎所有供应商的平台上，只要这个平台足够大。然而，想要充分利用新技术的优势，客户往往需要耗费巨资购买新设备。“第二个系统”的庞大规模实际上提升了各种硬件产品的销售量：更快的CPU、更大的硬盘、更高容量的磁带驱动器以及大量的内存芯片。这可给硬件厂商带来了巨大的利润。

因此，这“第二个系统”带给大家的感受可谓是喜忧参半。你得到了很多功能，有一些也许能用得上。而且你也有机会去说服你的老板，是时候购买新机器了。

3.5.4　“第二个系统”被大张旗鼓地誉为伟大的成就

“第二个系统”在市场上大展宏图。它的灵活性、多种选项和可扩展性为其在商业上获得了广泛好评。经销商们赞美其优秀品质：今天它就实现了未来的技术。在各个方面它都远超现有的系统。用户可谓是别无他求。

不明真相的群体有点儿不堪炒作重负，他们期待着专家的解答，而专家们也愿意为他们解答问题。设计委员会的任何成员（它现在已经发展到数百人）即刻披上了“专家的外衣”，备受人们尊重。其他人则煞有介事地评判设计委员会所做的工作，由此也获得了一呼百应的公信力。

人们对这个系统的兴趣有增无减。它成为了媒体的宠儿，各大杂志都蜂拥而上跟踪它的进度，揭秘其诀窍的书籍开始出现。人们召开各种会议让那些严肃认真的关注者探讨它的未来，研讨会帮助那些对其一知半解的人们探究它的历史。随着越来越多追随者参与进来，它俨然成为一股势不可挡的潮流。

一旦所有人都在吹捧这 “第二个系统”是一项杰出的成就，人们就会对此深信不疑，它便成为大家脑海中根深蒂固的思想。例如，X Window System吸收了许多超越其基本窗口功能的特性。尽管其中大部分功能几乎用不上，且它们的存在也极大地影响了系统的整体表现。但X Window System还是得以生存了下来，因为它毕竟是“第二个系统”。虽然它有无数缺点，但它还是强大到足以超越Unix市场上的其他任何窗口系统。它拥有至高无上的地位。而且，“第二个系统”是其他系统无可替代的——除掉“第三个系统”。

3.6　人类的“第三个系统”

3.6.1　“第三个系统”由那些为“第二个系统”所累的人们创建

经年累月之后，有些人开始发现，这“第二个系统”并不像它吹嘘的那样神乎其神。他们意识到它会吞噬系统的资源，运行起来也非常缓慢。它的确是由专家设计的，只不过其中一些人是自封的“砖家”。虽然“第二个系统”努力去满足每个人的需要，可实质上它并达不到任何人的要求。

不久，“第二个系统”便使许多人精疲力尽。他们认识到所谓的“令你别无他求的系统”只不过是众多系统的一个。它催生了太多的用户组，连处于技术领

域外围的人士都开始讲授研讨会。这些人并不像早期激发这股热潮的人们那么才华横溢，因此教学质量一路下滑。夸夸其谈充斥着这些用户会议，外围人士看到之前的专家早已声名鹊起，所以才希望能效仿这些人的派头。只是，他们来得太晚了。

人们都在窃窃私语地交换意见，也许这“第二个系统”并非那么完美，最后，这些小声的传言变成了广大用户发出的集体呐喊。每个人开始怀疑是否会有一个更好的解决办法。直到此时，全世界都做好了迎接“第三个系统”的准备。在与“第二个系统”的对抗中，“第三个系统”诞生了。

3.6.2 “第三个系统”通常会改变“第二个系统”的名称

当“第三个系统”到来的时候，“第一个系统”的发起人早已不知所踪。“第二个系统”开发工作中最有创意的那批人也都转向了更为有趣的项目。没有人想和这种日薄西山的技术联系在一起。

如果说，“第一个系统”过渡到“第二个系统”的转变过程就像是一次乘着新希望翅膀的翱翔之旅，那么，从“第二个系统”过渡到“第三个系统”则仿佛搭乘着泰坦尼克号：当这个金属庞然大物无法再漂浮在水面的时候，每个人都扑向了救生艇。

X Window System再次为我们提供了一个现实范例。X10是由主流供应商提供的第一款商业化产品。它具备“第一个系统”的典型特点：可接受的性能、几乎没有华而不实的特性、拥有激动人心的概念。而当X11出现的时候，它在各方面都体现了“第二个系统”的基本特性并将之发扬光大。最终“第三个系统”将会取代X11。在X Window System本身被视为过时系统的同时，它原来的名字也可能会逐步淡出人们的记忆。

3.6.3 最初的概念保持不变并显而易见

回想一下，“第一个系统”展示了一个能够激发人们想象力的概念。到“第三个系统”出现的时候，这个概念会被大家全盘理解和接受。每个使用过该系统的人都承认，它的做法恰到好处。“第一个系统”和“第二个系统”的开发思想依然存在于“第三个系统”里面。“火炼真金”之后，人们发现，它的确是真理。

最初的创新概念到今天成了司空见惯的常识，这样的例子不胜枚举。就拿墨水来说，曾几何时，人们用羽毛蘸着墨水在羊皮纸上书写。最终，钢笔取代了羽毛。然后近年来，（相对而言）圆珠笔成为了日常使用的标准书写工具。未来，还可能会有新型书写工具涌现，今天的圆珠笔会变成过时的工具。不过，假如你发现这些新工具使用的依然是墨水请不要惊讶。使用液体介质将我们的思想转移到纸张是承载文字的最初构想。墨水的发展经历了几种不同的容器，但只要我们还是使用纸张，墨水就会一直存在。

3.6.4　“第三个系统”结合了“第一个系统”和“第二个系统”的最佳特性

“第一个系统”在整体性能上有着上佳表现，但缺乏一些必要的功能；为了增加更多功能，“第二个系统”付出的代价却是性能不尽如人意；“第三个系统”达到了完美平衡，只保留了那些必需的功能。其实，只要有适当资源，系统就能实现大部分必需功能。

另外，“第三个系统”之所以拥有高性能还要归功于专家的贡献。他们可不是我们在前面讨论过的“伪专家”，而是真诚、积极且才华横溢的那些人，他们在这个系统的演变过程中做出了卓有成效的贡献。通过有意义的特定方式，他们付出的努力改善了系统。他们所做的工作，就像最初那个具有启发意义的思想一样，经得起时间考验。

在前两个系统中，企图将软件烧录在ROM中的举动是不可能成功的，这是因为软件系统的性质在不断发生变化。人们不应该把自己的“主观意见”放置在ROM中。然而，待到“第三个系统”出现的时候，软件设计师们对什么能成功，什么不能成功完全胸有成竹。时机已经成熟，将软件放置在硬件里面就具备了现实意义。

3.6.5　“第三个系统”的设计者有充裕的时间将任务做好

因为系统设计师采用的是成熟技术，所以人们易于了解这项任务的规模，它的风险也不大。决策者可以精确把握项目预算和系统实施的进度。

3.7　Linux既是“第三个系统”，又是“第二个系统”

Linux的出现正逢20世纪80年代末的动荡期，当时Unix处于嘈杂喧嚣的“第二

个系统”阶段。“第二个系统综合征”的迹象比比皆是，而Unix社区里，AT&T和伯克利这些主要派别正为哪一方实施的系统可以主宰Unix世界而争论不休。此外，人们还成立了诸如开放软件基金会（Open Software Foundation）的一些委员会，他们试图说服用户真正重要的是“开放接口”，而不是操作系统的哲学。这两种主流Unix社区推出的实施版本大多成了超级臃肿和缓慢的系统。

作为Unix哲学的下一个重要实现，而且是在一个几乎没有什么系统实施该哲学理念的领域，Linux展现出了许多“第三个系统”的特点。有些人厌倦了Unix世界中的自鸣得意和争吵不休，他们在Linux上面找到了很多乐趣。Unix中最好的原创概念在Linux下得以继续发展。许多Linux开发人员都抽出时间，帮助这个系统实现恰当的内核结构、接口和图形用户界面（GUI）。请大家不要忽略一个显而易见的事实：Linux其实就是Unix，只是换了个名字而已。

Linux自身也体现出一些“第二个系统”的特性。比如说，广为人知的“开源软件”系统其实是从Linux世界开始起步的。这个系统要经过一个漫长过程，在人们辩论、举办研讨会、并阐述其优缺点之后，才会进入“第三个系统”阶段。自此，永久自由分发该系统软件的源代码将成为普遍的做法。

3.8 建立“第三个系统”

我们的目标便是建立这“第三个系统”，从而让每个人付出的努力都能得到最大回报。它将功能、资源消耗和性能完美地结合在一起，里面只包括人们实际用得上的那些功能。它还在磁盘空间、内存和CPU周期这些硬件资源和系统性能之间取得了适当平衡。用户对它推崇备至，客户也年复一年购买着这个系统。

如何建立“第三个系统”？首先，你得先构建好之前的两个系统。除此之外，别无他法。任何企图改变此顺序的举动只是徒劳，你所建立的不过是更多的“第一个系统”或是“第二个系统”，这完全没有必要。

当然，其中也有一些捷径。秘诀就是尽快从“第一个系统”过渡到“第三个系统”。建立前两个系统的时间花得越长，就越久达到“第三个系统”最佳平衡状态。如果将构建“第一个系统”和“第二个系统”的工作周期尽量缩短，我们就能更快达成“第三个系统”。

尽管开发“第三个系统”是Unix开发人员和那些“传统”软件工程方法论秉承

者的共同目标，Unix开发人员却采取了一种更为激进的做法。首先，让我们来看看大多数软件的编写过程。

(1) 思考系统的设计。
(2) 建立一个原型来测试既定目标。
(3) 撰写详细的功能和设计规范。
(4) 编写代码。
(5) 测试软件。
(6) 修复现场测试过程中发现的bug和设计缺陷，并随时更新软件规格文档。

坚信事情无法一蹴而就的那些人发明了传统软件工程方法论。“如果你要建立一个系统，”他们说，“你同时也在建立‘第三个系统’。”但问题是，没有人知道什么是“第三个系统”，直到它被完成的一刻。

传统主义者喜欢记录所有要点，好像反复撰写规格文件就能保证他考虑到了所有的设计因素。之所以会出现这种“在编写第一行代码之前，就应该完成90%的设计工作”的观念，是因为人们认为深谋远虑才会产生优秀的设计。他们相信只要将设计思路记录成详实的文档，就可以保证自己已经调查过所有可行方案，而且还能借助完整的规格文件去理清所有思路，并最终提高工作效率。

这种传统方法导致的后果就是，人们在开发早期会拥有一份优秀的规格文件，可后期的文档质量却很差。在项目开展的同时，迫于进度的压力，工程师们的时间主要会用于编写软件，并减少撰写规格文件的时间。工作重点的转移导致规格文件无法与实际产品保持同步。如果有两个选择，一个是没什么说明文档的产品，另外一个是有详实的规格文件却完全没有产品实例，软件开发人员总是会选择前者。没有产品作为支撑，人们才不会为任何规格文件付钱呢。

Unix开发人员对待功能和设计文档有着不同的观点。虽然他们的意图与传统主义者类似，可执行步骤却有所不同。

(1) 撰写一份简短的功能规格文件。
(2) 编写软件。
(3) 使用迭代的软件测试/重新编写代码的开发过程，直至完成全部功能。
(4) 如果有必要的话，再撰写一份详细文档。

此处提到的简短功能规格文件通常是指3~4页，甚至更少。这样做的理由是：

(1) 没有人真正知道人们想要的功能是什么；(2) 人们很难去描述那些还不存在的事物。虽然传统主义者可能会费时费力撰写功能和设计规范，Unix程序员一开始却只是匆匆记下一些与目标直接相关的事情，然后就着手去构建系统。

那么，Unix程序员如何知道他们是在朝着正确的方向前进呢？其实他们并不清楚。传统主义者也不知道。最终，人们会将设计展示给有意向的最终用户。所不同的是，传统主义者呈现给用户的是一份大部头文档，里面包含着系统功能的枯燥说明，而Unix程序员却会向用户显示一个能初步运行的应用程序。

传统主义者不清楚最终的产品能不能满足用户需求。他也不能肯定用户是否行之有效地传递出自己的想法，同样，他不能肯定自己实施的系统是否符合这份规格文件。

与之相反，Unix阵营却提供了一个能实实在在跑起来的“第一个系统”。用户开始找到一些感觉，了解最终产品将如何运行。如果他喜欢这个系统，那很好。如果不喜欢，尽早调整也要比晚期伤筋动骨地修改更为容易。

请大家记住，“第一个系统”的特点是，它显示了一个能激发别人想象的概念。看到这“第一个系统”的鲜活实例用户的内心将点燃创意的火花，他开始想象该产品能完成的工作。这个火花越来越旺，他开始畅想新的用途，其中甚至会有一些连最初设计师都未曾考虑到的想法。

是的，Unix的方法要优于传统工程方法。在传统主义者的用户还不清楚产品是什么样子的时候，Unix开发人员的用户已经在考虑该如何处理这个工作原型了。

对于Unix用户，迭代的设计过程已经开始。他和开发人员正朝着“第三个系统”前进。一旦开发人员收到用户的初步反馈，他们就会知道自己的工作是否走向了正轨。有时，用户可能会告诉开发人员，这个产品并不是他想要的东西，那开发人员便可以重新开始设计。很多时候，用户喜欢该设计方案中的某些部分，并反馈建设性意见告诉开发人员哪些方面需要改变。这种开发人员和最终用户之间的协作，等同于在建立一个能够多方面满足用户需求的“第三个系统”。

撰写详细设计规格文件的最佳时期是在迭代设计完成之后。当然，到那时候也没必要写一份冗长的规格文档，因为它主要是供开发人员和感兴趣的第三方人员阅读和审查，而用户和开发人员早已在迭代设计过程中审阅过这个应用。如果由于某种原因仍然需要详细文档，那么对现有的应用程序进行扩充会比较简单。

有些人对软件开发中采用的Unix方法表示出顾虑，认为它虽然适合小型系统，对大型系统却不一定适用。他们认为，在没有详尽的设计方案之前就进入编码阶段无异于是“自杀之举”。他们还声称，缺乏足够的远见只会导致灾难性的后果。

我们并不是说，在没有适当设计之前，就应该投身到编写大型系统的任务中去。深思熟虑还是很有必要的，这可以让我们明确目标。不过，将过多细枝末节写在文档里面并没有太大必要，因为细节很可能会改变。关键是要确定该如何将系统解成小一些的组件。这样，每一个组件都能够在小的设计领域内完成它的架构。

我们还要考虑到，虽然Unix的一个好处就是在几乎没有什么前期设计的情况下，它已经发展成了一个强大的系统，能够执行那些曾经只能被大型系统处理的任务。传统上，具备类似规模的系统往往有着连篇累牍的大部头功能规格文档和详细设计文件。而大量现存的Unix文档却是在几个商业化技术出版社的督促之下才创建的，但其中大部分都是在描述已存在的设计。实际上，Unix并没有什么初始规格文件。

有别于大多数按照传统方式设计的系统，Unix系统是直接从原型演变而来的。它的发展经过了一系列迭代设计，将它从一个用途不大的实验室系统演变成一款能够解决最艰巨任务的系统。它是一种优秀设计哲学的鲜活体现，虽然对于某些人来说这并非是一种正统做法，但成果斐然。

第4章 可移植性的优先权

1991年的USENIX技术大会，我作为与会者漫步在纳什维尔的Opryland酒店大堂里，尽情欣赏着这座豪华的乡村音乐城堡。我穿行在拥挤的人群中，希望能够邂逅某个名人，然后炫耀一下自己刚刚签订了一份出版合同。

那时，我可谓是春风得意。出版社毫不犹豫地接受了我的提议，让我出一本关于Unix哲学的著作。（“很多人希望阅读这样的一本书。”）合同谈判的进展非常顺利。（“我希望先拿到*X*美元的订金。”“没问题，我们答应你的要求。”）交稿截止日期也很宽松。（“我们会额外多给你两个月，不想让你有太大的压力。”）是的，再没有比这更好的开局了。

说老实话，我却战战兢兢，心乱如麻。我得到了这份梦想的合同，这本书的创作题材我也渴望已久，可最初那股欣喜若狂的兴奋劲儿过去之后，我得直面现实，其实我并没有做好精神准备：见鬼，我真的要开始写书了吗？

我过去的写作经验只不过是为地方性娱乐杂志撰写一些专题文章。我学会了如何串词；我知道要注意主题句、实意动词的运用，以及被动语态的用法；我可以吸引并保持住读者的兴趣。但是，通过撰写杂志文章我成为了“一名不错的短跑选手”。而现在，我却要开始跑“马拉松”。

我第一时间飞奔到一个朋友那里去请求帮助。身为完成了几部大著的作家，他早已经历过类似的涅槃过程。我问他，应该怎么做才能处理好这项看似无法完成的任务？

“先去购买一台笔记本电脑。”他回答道。

看到我脸上疑惑不解的神情，他解释说，写书这件事情很特别，你要么无法下笔，要么思如泉涌。你需要精神高度集中才能将连绵不绝的想法倾注于在纸上。你必须时时刻刻考虑该如何创作这本书：早晨刷牙，开车上下班，会议间歇，吃午饭的时候，在健身俱乐部锻炼的时候，与家人一起看电视的时候，或是临睡之前。笔记本电脑是唯一那种适合随时输入文字的便携式设备，强大到足以让你在任何地方都能创作出一本书。

受益于高速发展的现代微系统技术，笔记本电脑的厚度通常还没有三孔活页夹厚，却能容纳下与桌面PC相当的计算能力。它的重量低于5磅[①]，紧凑的设计使得它像大学教材般便于携带。笔记本电脑拥有硬盘驱动器和内置的调制解调器，因此能够处理诸如电子表格的制作、文字处理、编程等日常计算任务。

几年前，苹果电脑公司曾经策划过一款电视广告，两名高管在讨论不同款个人电脑的优缺点。两个人都谈到了技术规格，其中一个人说的话饱含深意：最强大的计算机并不是那一款有着最快CPU、最大磁盘驱动器或是最强劲软件的机器。使用最频繁的那台计算机才最为强大。

如果按这个标准来判断，迟早有一天笔记本电脑会被列为有史以来最强大的电脑。虽然它不具备如实验室超级计算机那般快得令人眩目的处理能力，它也没有最新的磁盘存储容量技术，它的图形功能可能永远赶不上最先进的桌面显示屏。在它身上，你很难看到什么高性能机器的影子，但它有一个足以打动人的优势：便携性。

这带给我们下一条Unix哲学的准则[②]。你可能需要特别留意此条准则，因为它充分说明了Unix为何能成为软件领域的“常青树”。如果拿人的寿命来作类比，它可以说早就万寿无疆了。

4.1　准则4：舍高效率而取可移植性

软件开发过程涉及无数选择，每个选择都意味着各种妥协。有时，开发人员必须编写简短的子程序，因为他根本就没时间去编写复杂程序。其他时候，有限的内存可能是个制约因素。而在某些情况下，人们必须避免使用过多的小数据包来挤占

① 1磅≈0.45千克。——编者注

② “便携性”与“可移植性”的英文单词同为portability。——编者注

网络带宽，因为我们采用的网络协议对大数据块的传输更为有利。程序员总是要在一大堆方案中作出取舍，尽量去满足那些往往自相矛盾的目标。

其中，程序员要面对的一个艰难选择就是：高效率与可移植性。这也是一个极其折磨人的抉择，因为偏向高效率往往会导致代码不可移植，而选择可移植性却又会让软件的性能不那么尽如人意。

高效率软件不会浪费CPU周期。它充分利用了底层硬件的特性，可往往完全忽视了可移植性的问题。它利用了一些硬件功能，如图形加速器、高速缓冲存储器以及专门的浮点指令，等等。

虽然从纯粹主义者的立场而言，高效率的软件非常有吸引力。然而，可移植性意味着软件能够运行在许多不同的机器上，这使得人们考量的天平向可移植性这一端倾斜。这其中资金层面的因素要大于技术层面：在今天的计算环境中，那些只能运行在一种体系架构上的软件，其潜在市场竞争力会大打折扣。

当然，重视可移植性并不意味着你一定要开发出低效率、技术上晦涩难懂的软件。相反，从可移植性软件中获取最佳性能需要更高超的技术水平。如果水平不够，还有一个替代方案，那就是等到可用的硬件问世再动手。

4.1.1 下一……的硬件将会跑得更快

曾经我们在省略号处填入“年”：明年的硬件将会跑得更快。但由于当今硬件技术的迅猛发展，有的时候我们完全可以说，下个季度甚至下个月的硬件可能会跑得更快。如今，电脑制造商的产品开发周期日益缩短，他们只用花比过去短得多的时间就能推出新的型号。厂商们你追我赶，经常个个都号称自己的产品拥有更高性能和更低价格，借此与竞争对手一较高下。因此，不管你现在正在使用什么样的电脑，很快你就会觉得它又老又笨，就像当年大学实验室里的庞然大物一样。

硬件设计周期日益紧缩的趋势还在加剧。今天的半导体设计人员使用复杂的模拟器来构建升级版芯片。由于模拟器（它们自身就是一些强大的计算机）的运算速度持续提高，因此开发人员也就可以更快完成设计任务。然后，这些新的芯片便成为未来模拟器采用的动力引擎。一代代处理器如滚雪球一般发展。半导体设计世界的动向令人目不暇接，芯片的性能不断螺旋式加速上升。

随着高速机器逐步淘汰原来的慢机器，问题的关键已经不再是软件是否能充分利用硬件优势，而是软件能否运行在更新的机器上。你可能要花上几天或是几周的时间为今天平台上的一个应用调试优化出最佳性能，然而不久却发现，下次硬件升级会使得软件运行速度“自然而然地”快了10倍。不过，具备可移植性的软件才能利用下一款超级计算机的优势。

在Unix环境中，可移植性的含义通常意味着人们要转而采用shell脚本来编写软件。shell脚本由多个可执行的Unix命令构成，它们被放置在单独的文件里，可以间接由Unix的命令解释器来执行。因为典型Unix发布版本都会拥有一大堆小型而具备单一用途的命令，所以shell脚本几乎可以轻而易举地构建所有任务，最底层任务除外。

shell脚本附带的好处就是，比起C语言编写的程序，它们更具备可移植性。你应该只在别无他法的时候才考虑用C语言来编写程序，虽然这个想法可能听起来很另类。但在Unix环境中，C程序缺乏shell脚本的可移植性。C程序往往要依赖头文件、计算机体系结构，以及具体Unix版本中一系列不可移植的特性。在Unix从16位架构移植到32位和64位的同时，大量软件曾因为C语言相对较低的可移植性而无法使用。在这个方面，C语言只比20世纪80年代的汇编语言强那么一点点。

如果你想让自己的程序能在Unix之外的系统上运行，那么可以选择其他的语言，每一门编程语言都有自己独特的优缺点。如果你追求的是严格意义上的可移植性，那么Perl和Java便能较好地满足这一要求，它们在Unix和Windows平台上都能用。然而，将一门语言成功地移植到Unix之外的平台，并不意味着这个平台就能坚持Unix的哲学理念和工作方式。比如说，你会发现Windows上的Java开发人员往往是隐藏底层细节的互动式开发环境（interactive development environment，IDE）的忠实支持者，而Unix平台下的Java开发人员却喜欢能帮助自己深入研究图形用户界面内部细节的工具和环境。

4.1.2　不要花太多时间去优化程序

如果程序运行的速度还算可以，那么就接受事实，承认它已经满足了你的需求。在琢磨着优化子程序和消除关键瓶颈的同时，我们还应该思考如何能在未来的硬件平台上提升性能。不要单纯为了求快而优化软件。请记住，明年的机器很快就要问世了。

很多Unix程序员常犯的错误就是，为了获得性能上一些微不足道的优势而采用C语言去重写shell脚本。这纯粹是浪费时间，我们完全可以把这些时间用在获取用户的建设性回应上。也许有时候shell脚本的运行速度真的不够快。不过，就算你确实需要较高的性能，而且坚信只有C语言才能达到效果，还是得三思而后行。尽管程序确实会有得满足特殊需求的时候，但通常情况下还是没必要用C语言来重写脚本。

请关注“微优化”（micro-optimizations）这个概念。如果（而且这个“如果”是不容忽视的）必须要优化C程序的性能，Unix平台提供了`prof`和其他一些工具来定位使用得最频繁的子程序。对这些被调用过成百甚至数千次的子程序进行优化，可以产生事半功倍的效果。

另一种提高程序性能的方法就是研究如何处理数据。例如，我曾写过一个简单的shell脚本，可以在不到一秒的时间内搜索分散在几千个文件中的二百多万行代码。我的诀窍就是先建立一个数据索引。每个索引行包括一个单独的词以及所有包含这个词的文件列表。大多数程序使用的字符不会超过20 000个，这意味着索引数将不会超过20 000行。对于Unix工具`grep`来说，查找20 000行的文本是一个相对简单的任务。一旦`grep`定位到这个单独的词条，它便会打印出与该词条相关的文件名列表。它的查找速度非常快，因为它采用的方式是提前查阅好数据，这也使得这个程序具有了可移植性。

4.1.3 最高效的方法通常不可移植

任何时候，但凡一个程序利用了某些特殊硬件功能的优势，它在变得更高效的同时，却也不那么可移植了。特殊功能有时候会极大改善软件的性能，但它们需要采用与硬件设备相关的代码，当目标硬件被更快的版本取代之后，人们就需要更新代码。虽然更新与特定硬件相关的代码为系统程序员提供了许多“就业保障”，可雇主却得不到什么好处。

在我的职业生涯中，我加入过的一个设计团队要为一款新硬件平台创建一版早期的X Window System。这期间有位工程师编写了几个演示程序，绕过X Window System并应用了该硬件的先进图形功能。另一位工程师也编写了一套类似的演示程序，不过它构建在X Window System提供的可移植性接口之上。第一个工程师的演示效果令人赞不绝口，因为他使用了最先进的图形加速器，拥有令

人难以置信的高效性能并将硬件特性发挥到极致。相反第二个工程师的演示程序的运行速度就较慢一些，但因为运行于X Window System之上，它却扎扎实实地保持了可移植性。

最终，曾经最先进的图形硬件变得不那么先进，而且公司开发出了一款更快但完全不向前兼容的新型图形芯片组。在新硬件上重新实现第一个工程师的演示程序需要付出艰苦卓绝的努力，所有人都没那么多时间。因此，第一个演示程序默默无闻地消失了。而另外一个能够运行在X Window System上的演示程序具备可移植性，不需要做什么额外修改工作就被移植到了新系统，在我撰写本文的时候，它可能仍然在使用中。

当你出于效率考量利用到一些特定硬件的功能时，你的软件就成为兜售该硬件的工具，而失去了软件自身的立场。这限制了它作为一个软件产品的能力，而它的销售价格也实现不了其内在价值。

那些与硬件平台紧密结合的软件只在该硬件平台具有竞争力的时候才能维持它的价值。一旦该平台的优势消失殆尽，软件的价值就会大大地下降。为了保值，它必须从一个平台移植到另一个可用的、更新更快的架构。如果不能迅速采取行动转移到下一个可用的硬件平台，那对软件而言就意味着命运的终结。市场的机会之门在关闭之前，只会短暂地开放。如果软件赶不上这个时间窗，它的市场地位就会被竞争对手取代。有人甚至认为，无法将自己的软件升级到最新平台是软件企业倒闭的首要原因，超过了其他所有原因的总和。

衡量应用程序是否成功的一个标准就是它能够在多少个系统上运行。显然，与一款备受市场瞩目且可以运行在多个供应商系统上的应用相比，依赖于单一供应商硬件平台的程序将难以成为前者的主要竞争对手。从本质上讲，可移植的软件比高效率的软件更有价值。

在转移到一个新平台的时候，充分考虑了可移植性因素的软件大大降低了它的平台转移成本。由于开发人员不需要耗费太多时间在移植工作上，因此他便可以把更多时间用来开发新功能，由此吸引到更多用户，同时也赋予产品新的商业优势。因此，从它诞生的那天开始，可移植的软件就更有价值。人们在早期为可移植性付出的所有努力都会在后期得到丰厚回报。

4.1.4 可移植的软件还减少了用户培训的需求

在用户花时间学会了该如何使用某个应用程序组件之后，如果未来平台上还运行这款软件组件，他之前的辛苦就会得到回报。该软件的未来版本可能会略有变化，但支撑它的核心理念和用户接口很可能始终如一。在产品每次升级的过程中，用户对该产品的体验也得到了增强，随着时间的推移，他就会由新手用户变成高级用户。

从Unix的最早版本开始，这种从新手用户转变为高级用户的过程一直在持续进行。人们在早期版本中学到的大部分知识仍然直接适用于今天的Linux平台。诚然，Linux的确添加了一些自己的新特性，但总的来说，由于整体用户环境（从shell到常用工具）都具备可移植性，因此那些经验丰富的Unix用户在Linux环境中还是能游刃有余。

4.1.5 好程序永不会消失，而会被移植到新平台

你有没有注意到，有些程序以这样那样的形式一直存在着？人们总是觉得它们特别好用。它们拥有真正的内在价值。总会有人出于好玩或是谋取利润等目的，主动承担起编写或将它们移植到当前流行硬件平台上的使命。

就拿Emacs风格的文本编辑器来说。虽然在某些方面，它们是Unix体系中的反面典型，但它们一直都是广大程序员和Unix爱好者的最爱。不单Unix，你还能在其他系统中找到它的变种。虽然多年来，有些版本的Emacs早已发展为繁琐的庞然大物，可在它的最简版本中，Emacs还是提供了一个不错的“无模式”手段，用于文本的输入和编辑。

另外一个好例子就是微软的Office和类似产品中包含的那些典型商业程序。人们发现，为了在现代商业环境中高效工作，大家总是需要像文字处理器、电子表格和演示工具这样的桌面程序。

然而，没有任何个人、公司、组织甚至于国家能够将一个不错的主意占为己有。最终，其他人都会注意到这个想法，并且开始制作所谓的“合理复制版本”。聪明的群体能够意识到它的价值，他们会努力在尽可能多的平台上实现这个想法，并尽可能多地抢占市场份额。实现这一目标的最有效方式就是编写可移植的软件。

案例研究：雅达利2600 游戏机

让我们来研究一下雅达利2600游戏机（Atari 2600），或者也可以叫它雅达利视频计算机系统（Atari Video Computer System，VCS）。VCS是史上第一款成功的家庭视频游戏系统，可谓是天时地利之作。它成功地迎合了人们的想象力，当时大家刚刚从本地酒吧和电玩城那里浅尝到Space Invaders（太空入侵者）游戏的滋味，并准备好了将这个视频游戏的新世界带入自己家中。它是第一款盒式－可编程的游戏机，改变了整个游戏行业的格局，并给人们带来了莫大的游戏乐趣，这些曾经只藏在校园实验室和软件专业人士的隐藏目录里的游戏，被引入普通家庭的电视中。如果说程序员有着与生俱来对游戏的向往，那这种热情与美国乃至全世界人群喜好玩游戏而产生的巨大利润相比，可以说是微不足道。

雅达利2600在问世之初就有着颇高的性价比，它的表现不错，大家付出的真金白银物有所值。当时，游戏机的售价约为100美元。它配备了两个游戏操纵杆和一对桨状控制手柄。 雅达利还在其中捆绑了一个名为“战斗”的游戏盒带，它包括各种结合了坦克、喷气式飞机以及福克飞机的两人战斗游戏。

雅达利并没有从游戏控制台的销售量中挣到什么钱，庞大的利润来自游戏盒带。这些盒带每盒售价为14美元到35美元不等，它们成为雅达利的主要收入来源，同时，大量小型软件公司也希望能从电子游戏热潮中分得一杯羹。从雅达利的运营角度来看，一旦它收回了开发游戏盒带的投资成本，其余的都是纯利润。

我有个朋友就在一家为雅达利2600生产游戏盒带的软件公司谋得一职。他解释说，将一个象棋游戏或是“射击”游戏挤进只有不到8K ROM的机器上可真不容易。这相当于在大众甲壳虫汽车里塞满20个人，不是每个人都有机会能看到窗外的风景。

在这段为游戏盒带编写代码的职业生涯中，他编写了一些最高效但不具备可移植性的软件。他把指令视为数据，数据当成指令。在执行水平回扫动作时，也就是电视机上的光束在扫完屏幕右边的最后一个光点再返回到屏幕左边的时候，软件还采用了某些特殊操作。为了节省内存空间，他绞尽脑汁用上了每一个可能的捷径。他的代码看上去很美，却是软件维护者的噩梦。

在2600产品发售期内，雅达利公司还推出了一款以6502为基础的800系统，它成为家用电脑产品线的旗舰产品。800系统是第一款真正意义上的计算机，因为它有一个打字机式的键盘，以及二级存储和通信设备的接口。它的售价将近1000美元，对2600主导的市场也没有造成很大的威胁，可这份好运只持续到内存芯片降价之前。

由于其他供应商的竞争和800电脑深受欢迎的图形扩展功能，雅达利迫切需要推出一款面向大众市场，能够运行800电脑软件的视频游戏机。这款新机器被

称为5200，它的出现使得大众市场的电脑盲们也可以运行技术人员在800电脑上玩的同款游戏。

一旦大众市场发现了这款魅力无穷的新机器能够提供更为平滑的图形界面，2600系统便被人们毫不犹豫地抛弃。随后雅达利2600游戏盒带价格立即下跌到谷底。预料到2600寿终正寝命运的经销商们开始降价销售他们所拥有的库存产品，市场上充斥着甩卖的2600盒带。这进一步促使它的价格暴跌，同时也波及了很多软件公司。

对盒带生产商而言，苦难并没有结束。虽然大部分在2600上流行的游戏也成为5200的热门，但在此之前，它们必须经过重新编写才能运行在新的硬件平台上。由于2600盒带上的代码过于高效，几乎没有丝毫的可移植性。这意味着人们需要花费巨大代价对这些软件进行改写。

问题是，虽然这些游戏盒带软件可谓说是有史以来最高效的产品，但在新硬件问世的当天，它的价值却一落千丈，这都是因为它不具备可移植性，无法重新编译并在5200上重用。如果这些代码是可移植的，也许可以改写视频游戏产业的历史。雅达利公司很有可能成为世界上最大的软件供应商。

最后请注意，今天你可能只需要花个几美分便能买到一个雅达利2600的游戏盒带，不过就算你会去买，我估计也只是怀旧的成分居多。与此同时，为了购买微软Office功能最强大的版本，你花的钱不会少于几百美金。有部分原因就是，在英特尔公司发布更强大的8086处理器版本时，微软Office都能从旧平台迁移到新平台。这虽然让Office程序一直保持在技术领域的前沿，可却要付出很高的代价。

然而，Office的开发人员必须时刻保持警惕，密切关注未来Open-Office的动向和其他出现在Linux市场中的Office克隆产品。其中一些产品的内在可移植性要远远高于Office，这可能意味着未来它们能够更加适应商业技术的发展演变。微软也许得耗费巨大财力物力才能维护Office的霸主地位。只要英特尔继续生产向后兼容指令集的CPU芯片，微软的Office组件（从这个意义而言，微软Windows本身也一样）的移植工作就不会太难。但如果有人设计出了一款非常先进的机器，每个人都希望拥有，哪怕它有着与英特尔毫不兼容的结构，那微软就将面临庞大的工作量，需要大费周章将Windows和Office移植到新的架构，否则就只能坐以待毙。

这个故事的寓意是什么？可移植性有着巨大回报。其他关于效率方面的考量不过是小问题。

迄今为止，我们一直在对高效软件和可移植性软件作比较。能够轻松移到新平台的代码远比那些需要利用硬件专属特性的代码更具有优势。我们已经看到，完全可以用实际标准（即金钱）来判定这个原理的重要性。为了保住自己的利润根基，软件公司应该努力实现产品的可移植性，并在这个过程中力争提高效率。

然而，对于可移植性这一目标而言，可移植的代码只完成了一半工作。所有应用程序都由指令和数据构成。指令的可移植性可以确保代码还能运行在明年的机器上。那么数据呢？置之不理吗？当然不行。Unix程序员不仅得编写可移植代码，数据也一样。

怎样才能让数据具备可移植性呢？ Unix哲学的下一条准则提供了一个解决方案。

4.2 准则5：采用纯文本文件来存储数据

“纯文本文件”的意思是你必须将所有数据都存储为文本，就这么简单。将文件保存为二进制格式是严格禁止的。任何特殊文件系统的格式都不行。这便将那些供应商出于某些冠冕堂皇的目的而创造出的不可移植的格式排除在外。用Unix世界的行话来说，数据文件应该只包括一组由换行字符或“newline”分隔的字节流。

许多人认为这种做法是副难以下咽的苦药，但Unix程序员都坚信它的效果最好。秘密就在于：尽管我们可以将数据保存在任何存储介质上，但它最终还是要去往某个地方。驻留在磁盘上的数据没有价值。想让数据保持活力并拥有价值，它就必须时不时地移动。否则，人们应该将它们打包归档并删掉。

无处可去的数据就成了“死”数据。

如果你想轻松移动数据，就必须使它具备可移植性。任何阻碍数据移动的因素，不管是无意还是经过精心设计的，都降低了这些数据的潜在价值。数据驻留在一个地方越久，在它到达最终目的地时，其价值就会越小。问题是，如果数据没有与目的地系统匹配的格式，必须被转换。这个转换过程需要时间。花费在数据转换过程中的每分每秒都在蚕食着这些数据的价值。

CNN（美国有线电视新闻网）曾因为它在1991年波斯湾战争报道中的杰出表现而获得最高荣誉。CNN迅速为全世界人们生动呈现了冲突的现场画面。每天晚上，许多人都赶回家中，守着电视机观看事态发展。如果制作人员需要花费数天时间将视频从beta格式转换为VHS格式再空邮到亚特兰大，并且只在黄金时间播出的话，那CNN的报道还会这么引人入胜吗？

数据的可移植性也是如此。如果我们先得大费周章来转换数据格式，然后才能

将移动它的话，那这些数据在到达目的地时，其价值就会降低。这个世界节奏如此之快，没有时间去等着你转换数据。

4.2.1 文本是通用的可转换格式

文本不一定是性能最好的格式，它只是最通用的。某些应用程序采用的格式都不像文本文件这般有着广泛的接受度。几乎所有的目标平台都能处理文本编码的数据。

使用文本可以消除很多麻烦，你不需要将自己的数据从一种二进制格式转换到另一种二进制格式。很少有标准化的二进制格式。每个供应商都定义了自己的二进制编码，而且大部分互不兼容。从一个供应商的格式转换到其他格式会是一项艰巨任务，耗时从几天到几个月不等。这些时间如果能花在数据的使用上，效果会更好。

4.2.2 文本文件易于阅读和编辑

人们可以轻松查阅文本数据，不需要使用任何转换工具。如果数据有误，你还可以用标准文本编辑器对它加以修改。你不需要专有工具，也不需要为每个单独的数据文件准备不同类型的编辑器，“一招鲜，吃遍天”。

在开发使用Unix管道（pipe）的程序时，文本文件真正的力量开始显现。管道是一种无需使用临时文件，直接将一个程序输出作为另外一个程序输入的机制。许多Unix程序只不过是由一系列管道连接在一起的一系列小程序集合。开发人员创建软件原型的时候，他们可以日益挑选管道沿线节点来检查数据的准确性。如果有问题，他们可以中断通过管道的数据流，并找出出错的到底是数据还是操纵数据的机制。这大大加快了开发进程，使Unix程序员比其他操作系统的程序员拥有明显优势。

文本文件还简化了Unix用户与系统的接口。大多数Unix下的管理信息都保存在纯文本文件里，以供公众用户查阅。这大大减少了个人用于获取这些信息来完成日常工作所花费的时间。人们可以不费吹灰之力收集到关于其他用户、网络上的系统以及一般统计数据的信息。讽刺的是，在这里可移植的数据反而提高了工作效率。

4.2.3 文本数据文件简化了 Unix 文本工具的使用

大多数Unix环境都包括几十个传送、修改和过滤文字的实用工具。Unix用户运

用多种方式将这些工具加以组合来完成他们的日常工作。表4-1列举了一些比较广受欢迎的工具，并简要说明了其功能。

表4-1　Unix中的常用工具及其功能说明

工　具　名	功能说明
awk	对以字段组织的文本进行操作
cut	从文本中提取特殊的文本列
diff	逐行比较两个文本文件
expand	将制表符转换成空格
expr	从一个字符串中提取部分字符
fmt	一个简单的段落格式化工具
grep	在文本文件中搜索指定字符串，并显示匹配文本行
head	显示文件的前*n*行
lex	对文件流进行词法分析
more	每一次都只显示单屏文件内容
paste	将单列文本拆分成多列
roff	综合性文本格式化和排版设置工具
sed	一款非互动形式的文本编辑器
sort	对文本列作排序
tail	显示文件内容的最后*n*行
test	比较两个字符串是否相同
tr	替换文本中选定的字符
wc	计算文件的行数、字数和字节数

这些工具还拥有很多我在这里没有提到的功能。比如，awk命令可以混合使用字符形式的文本和数字形式的文本，并在它们之间作相互转换。test命令可以用于检查文件的模态，以了解它们是否可写。lex命令通过匹配输入流中的字符串表达式来提供一个对应C语言的接口。sed本身就是一个强大工具，足以取代如grep、head和tail这样的命令。

这些命令的混合模式功能往往会模糊了文本和传统数据之间的界限。因此，那些以前存储在二进制文件中的数据更容易以文本形式呈现。Unix程序员经常在文本文件中存储数值数据，因为Unix环境提供了丰富的工具来处理这些文件。

将数据存储为文本形式，然后使用不同的面向文本的小型工具来对数据进行操作，这使得Unix成为令人惊叹的数据处理器。有了诸如awk、sed和grep这样在几

乎所有Linux和Unix系统上都能找到的工具，人们便可以轻而易举地选择、修改和移动数据。即便普通人都会发现阅读和解释纯文本文件中存储的数据更为轻松。

惠普OpenVMS[①]操作系统的开发人员认为大多数人都害怕使用计算机，他们的想法可能没错。然而，Unix的做法却恰恰相反，它们非但没有将用户屏蔽在系统之外，反而将他们带入到Unix的内部世界。它引导用户穿越错综复杂的逻辑路径，而且他们仍可以坚守自己熟悉的旧日痕迹，也就是说他们的数据格式易于阅读和理解。尽管人们喜欢发表诸如“Unix用户界面不友好”这样的批评意见，但其实Unix可能是一款最友好的系统。用户不必成为一位能够解释复杂二进制文件格式的高级技术大师，就可以随时查看他们的数据。

4.2.4 可移植性的提高克服了速度的不足

在整个讨论中，你可能一直在想：“嗯，可移植性是不错，但系统的整体表现怎么办？”是的，纯文本文件的使用确实拖累了系统的处理速度。一个二进制字节的内容需要用两个或三个字符来表示，所以性能降低了差不多三分之二。这听起来似乎很严重，但实际情况却并非如此，除非这是一个高分辨率的实时应用程序或是罕见的多TB数据仓库应用程序。然而，即使在一个庞大的数据仓库应用程序下，用户能够吸收和分析的数据量还是不值一提（按照人类的标准）。因此，在某些时候，使用二进制格式存储而节省的数据量对于CPU处理速度的周期标准来说，其实是微不足道的。

每一个应用程序最终都会被移植到新的系统，否则就会不复存在。电脑制造商在持续不断地进步，这完全保证了今天的昂贵机器可能会在明天变得异常便宜。在一个缓慢的，而且维护代价日益昂贵的系统上运行应用程序并不是划算。

当你需要将应用程序移植到新架构时，使用文本存储数据的做法就有了回报。如果你之前有足够远见让程序具备可移植性，那么采用文本存储数据也会减轻很多负担，将数据转移到新平台就变得轻而易举。那些需要同时移植代码和数据的软件工程师可就没那么轻松。在新内存条问世的时候，这些数据就会过时。由于性能降低三分之二而浪费掉的那点儿时间，与将数据移动到新平台而需要花费的数周甚至数月相比，可谓是不值一提。

① OpenVMS系统最早由DEC公司开发。DEC公司后来被康柏公司收购，康柏公司则被惠普公司收购。

4.2.5　速度欠佳的缺点会被明年的机器克服

我们承认文本文件的确拖累了系统的性能。程序运行速度最高可能下降三分之二。但是，如果应用程序达到了今天的最低性能要求，那么在数据可以被移植到新机器的前提下，我们能预见到明年的机器必将有大幅改善。

如前所述，明年的机器通常能够额外提供足够的计算能力，于是，今天我们关于文本文件影响系统性能的担忧就变得有点儿多余。换句话说，如果应用程序能在今天表现得中规中矩，那明天，它的运行速度必将显著提高。在未来几年内，你甚至可能要开始考虑该如何将运行速度放慢以适应人们的操作节奏。

案例分析：Unix大师的工具锦囊

我们已经看到，如果要在高效率和高可移植性两者之间作出抉择，Unix程序员很明显地青睐后者。因此，他们的应用程序往往会是第一个在可用新平台上运行的程序。这使他们的软件在市场上占据着绝对优势。在当今世界，机会之门会在一夜之间开启，然后一个月之后便猛然关闭，因而人们对于可移植性的追求可能会带来天渊之别，你要么成为一个行业领先者，要么就只能成为艳羡不已的旁观者。

Unix程序员是怎么逐步接受这些理念的呢？早期，大多数软件工程师并没有在学校里学到可移植性的重要性，至少没怎么听说过肯定这一做法的确切信息。那么最可能的情况就是，他们通过最好的方式，也就是第一手的经验了解到可移植代码与数据的价值所在。

大多数被称为Unix“大师”的人都会拥有自己的常用工具集，里面包括一些程序和shell脚本。在他们从一台机器换到另一台机器，一份工作换到另一份工作，一家公司到另一家公司的同时，这些工具都如影相随。为了详加说明，让我们来看看Unix大师的锦囊里都会有些什么宝贝。

这些年来，我个人收藏的工具一直都在变化。表4-2列出了部分具有代表性且经受住了时间和可移植性考验的工具。

表4-2　部分收藏工具列表

工具名	描　述
`cal`	Unix `cal`命令的前台shell脚本，它允许用户采用文本而不是数字来指代月份。引自Brian Kernighan和Rob Pike所著的*The Unix Programming Environment*[①]一书

① 此书出版于1984年，贝尔电话实验室公司拥有此书版权。

（续）

工具名	描　述
cp	它是Unix cp程序的前端版本，防止用户在无意中覆盖现有文件
l	打开 -F选项来运行 ls 命令
ll	打开 -l选项来运行 ls 命令
mv	它有点类似于cp脚本。它会防止用户因使用已存在的文件名来为新文件重新命名而覆盖掉原来那个文件
vit	为了使用标签和带标签的文件，在调用vi编辑器时打开 -t 选项。标签有助于在文件集中轻而易举地定位到子程序

我已经为一些脚本取好了别名，以方便在C shell中使用它们。C shell是一款交互式的命令解释器，最早出现在伯克利Unix系统上。别名允许你为自己惯常使用的命令指定一个特有的替代名称，这样就无需在shell脚本中键入它原有的长名称。就像shell脚本一样，它们也都具有可移植性。

我最早是在一家开发报纸排版系统的小公司建立起这个工具集，当时使用的是DEC公司的PDP-11/70机器，Unix的版本是第7版。由于公司新添了一些用于软件开发的新系统，因此我将这些工具移到同样运行着Unix操作系统的PDP-11/34、PDP-11/44和LSI-11/23机器上。虽然考虑到PDP-11产品线的兼容性众所周知，我这些动作听起来并不像是什么壮举。别急，后面的故事更精彩。

最后，我离开了这家公司，去寻求其他的职业发展机会。我把这些工具存放在一盘九轨道磁带上。很快，我这些C程序和shell脚本就找到了它们的新家：DEC公司的VAX-11/750机器。与我曾经使用过的小PDP-11机器相比，这款VAX-11/750机器的马力更强。因此，在新公司，这些工具的运行速度快了一点儿。在公司把VAX-11/750机器淘汰掉并升级成VAX-11/780之后，这些工具跑得更快了。而我并没有对它们做过任何修改。

正在那时候，工作站（这些奇妙无比的机器，其性能几乎可以秒杀所有其他机器）出现在人们的视线中。每个人都蜂拥而上，抢购这款由Sun公司出品的热门新机器，包括我的雇主。然后，那些从PDP-11移植到VAX产品线的工具不需要任何修改便能在Sun工作站上运行了。

在新英格兰地区度过了我软件工程生涯的大部分时光之后，我发现DEC公司出品的最新设备在波士顿160公里的范围内已经相当普及。同样，我把那些老而可靠的C程序和shell脚本移植到了最新的DEC产品线，这一次是VAX 8600系列，后来又移植到VAX 8800系列。同样，我还是不需要对这些工具作任何修改。

需求是发明的驱动力。和我共事的一名软件工程师留意到仓库里闲置着一大堆Digital Professional 350机器。我这位颇有头脑的同事认为，这些350机器可以当做我们的家用电脑，如果它们运行的是Unix系统那就更好了。于是，他着手将Unix系统移植到350机器上。我的工具紧随其后。

接着便是VAXstations和X Window System的早期版本。可移植的窗口系统是用户界面发展的重要一步。尽管窗口系统有着杰出表现，可我觉得自己还是最喜欢在xterm（一个终端仿真器）窗口下使用我的工具。

但计算机产业是一项高风险产业。想屹立行业之巅就必须极具灵活性。对于软件工程师来说，灵活性其实指的就是可移植性。如果软件不能移植到最新的机器，它就无法存活。所以，当基于RISC架构的 DECstation 3100系列和5000系列问世时，就出现了一个要么移植要么死亡的严峻局面。我的工具再次表现出具备可移植性的优越之处。

今天，我的小工具在各式各样大小不一的机器上运行，无需修改就能跑在各种Linux发行版下。

这些C程序和shell脚本已经正常使用了二十多个年头，在不同厂商的设备上，在不同的Unix版本下，从PC到小型机再到大型机，涵盖了所有从16位到32位再到64位的CPU架构。你还知道有其他什么程序可以用于这么多环境、使用这么多年吗？

这样的经历并不少见。在世界各地，发生在Linux和Unix程序员身上的类似故事不胜枚举。几乎每个经常使用Linux或Unix的人都有着自己的工具收藏。毫无疑问，有些人的工具包比我的还全面。其他人则很有可能将他们的软件移植到了更多的平台，所有的软件基本都不用怎么改动，用户也不需要接受额外培训。

可移植性的记录可以说明一切。在让程序和数据变得易于移植的时候，你就为软件创造了一种持久的有实际意义的价值。就这么简单。那些选择高效率的代码和数据只会将自己捆绑在早先的架构上。随着新平台不断涌现，不可移植的软件只能接受被废弃的命运。如果你不想看到自己心血随着新行业标准的公布而付之东流，就要提前作好准备。设计软件要时刻谨记让它们具有可移植性。

某一天，当你拥有第一台100 PHz（petahertz）主频、500艾字节（exabytes）存储空间[①]的笔记本电脑时（它的问世可能比你想象的还要快），要确保你的软件已为此作好了准备。

① PHz相当于一万亿（10^{15}）赫兹。艾字节约为10^{18}字节或大约一亿GB字节。你还要考虑到明年的新型号将会运行得更快！

第5章 软件的杠杆效应

5

要繁衍增多。

——创世纪1:28

如果你想致富，那就去卖特百惠（Tupperware）吧。这些塑料储藏盒是用来将食物密封保鲜的容器，你可以用一大堆没有标签的盒子塞满你的冰箱。这些保鲜盒独特的魅力使得它们早已成为家庭必需品，你还会希望自己另外有一个6英尺[①]高的厨柜来容纳它们。每个家庭的角落里或多或少都塞着几个特百惠保鲜盒。

个体经销商通过“聚会计划”来销售特百惠：他们在一些赞助人的家中召集一些非正式聚会，并给予这些赞助人一些回扣。赞助人在聚会上非常忙碌，他们要做产品演示，提供使用技巧，并接受订单。这是一份苦差事。其中一些人甚至只能挣得一点儿蝇头小利。

我的姑姑却在一年时间内，卖出了价值将近一百万美元的特百惠产品。

当我听说这件事情时，我首先想到的就是：“这得卖出多少保鲜盒啊!”在慢慢适应了我们家族中很快就会出现一个新百万富翁这一震惊的消息之后，我开始思索她是如何成功的。她只是一个普通人，家族中普普通通的一份子。她并不是那种你觉得能够发家致富的人才。

我知道她很努力，她似乎有无穷无尽的能量，不断攀升到自己事业的新高峰。在每一个到访的地方，她都侃侃而谈，给人们讲述特百惠是多么好的产品。就算是

① 1英尺=0.304 8米。——编者注

她来我们家度假的时候，都会辛勤工作到深夜，抓紧完成她的业务文书工作。

不过，尽管她的确付出了艰苦卓绝的努力，可我还是很难将她与百万富翁划上等号，算来算去也觉得不太对。我们且假设她出售的每个特百惠容器平均价格为7美元。为了达到一百万美元销售额，她大概需要出售142 858个盒子。我们再假设她每周工作6天，一年工作50周，这样的话她每天都得卖掉477个盒子。

现在，我的姑姑已经是一位了不起的推销员。她的销售业绩成为了传奇。凡是能卖的东西她都能成功地让你买下来。但一天要出售477个盒子，而且是每天（星期日除外），还是令人难以想象。毕竟，人是要休息的。而且，她平常还要照顾家里。

所以有一天我把她拉到一旁，问她是怎么做到的，居然能在一年内卖出价值一百万美元的特百惠产品。她的回答竟然是："傻瓜！我自己并没有卖出那么多盒子。我都是让别人来帮我卖的！"

她解释道，一开始她都是在晚上的家庭聚会上出售特百惠产品。她算了算，顺利的话她一星期可以举办5次聚会，每次聚会的销售额大概是100~150美元之间。最终她意识到，虽然她是位不错的推销员，但一个人的精力毕竟有限。所以她找来了20个人，对他们进行游说，说服他们也来销售特百惠。每个人都可以在一周内举办5次聚会，那总的算来每星期就能有100次聚会。她将她的盒子以薄利卖给他们。不久之后，这20人都明白了"金字塔"营销的价值。很快，其中一些人也发展了20个下线，来帮他们自己销售特百惠产品。

接下来的就是一个多层次市场营销的故事。

然后我的姑姑和我分享了一条人生哲理：无论你有多么聪明过人、精力充沛或是锐意进取，在人生的漫漫长途中，一个人的精力就只有这么多。如果想取得非凡成就，你就必须放大自己对这个世界的影响力。就算你智商很高或是能力强到可以在夏威夷出售冬季的皮大衣，也远远不够。你需要创办一些运营良好的业务实体，才能广为传播自己的才华和能力，并提高自己对外界的影响力。对于你工作的每一个小时，你都应该期望自己的努力能被放大十倍、一百倍，甚至是一千倍。

这其中的关键词是"杠杆"。就如同你在高中物理课上学过的杠杆和支点的原理一样，在杠杆任何一端产生的作用力都会传递到另一端。如果支点恰好在杠杆的正中央，那两端就会产生对等的相互作用。如果支点朝着一端向上移动5个单位，另一端就会下降5个单位。不过，如果你站在杠杆的一端，并将支点尽量靠近你自

己，那发生在你这一端的微小作用力最终都会在另一端产生巨大的作用效果。你的目标就是要寻找一种方法，努力使得支点能够更加靠近自己。换句话说，你要努力达到这样的效果，在杠杆这一端稍稍动一下，却能使另一端撬动月球。

为什么Unix会越来越成功，原因之一就是它能够为个体的工作发挥出巨大的杠杆效应。它的成功并非是偶然现象，而是很多人共同努力的结果。早期只是几十个人参加了这种协同设计工作，后来开始有上百个程序员加入这个团队。这些人意识到，如果仅凭一己之力，能做的事情只有那么多。但是，如果他们可以利用软件的杠杆优势，他们就能成倍放大自己的影响力。

5.1 准则 6：充分利用软件的杠杆效应

让我们假设你是世界上最好的程序员之一。自己编写的每一行代码都具有点石成金的能力。写的应用程序在发布当天立刻就会成为人们追捧的对象。各大网站的评论员对你的杰作好评如潮，你的软件就像是交易会上的明珠。它是真正独一无二的杰作。

不幸的是，成为“独一无二”的杰作会带来一个问题。这份独特性反而会成为牵绊你的枷锁。如果所有的工作都亲力亲为，其实你能做的事情也只有那么多。除非你能想办法来让别人来分担一些，否则在发挥出最大潜能之前，你会耗尽自己的全部能量。

5.1.1 良好的程序员编写优秀代码，优秀的程序员借用优秀代码

想编写大量软件，最好的方法就是借用别人的成果。这里所说的“借用别人的成果”是指将他人的软件模块、程序和配置文件集成到自己的应用程序中。通过制作衍生产品，你成倍放大了前任开发人员的努力，将他们的成果发扬光大到新的高度。他们的软件变得更有价值，因为它们的身影出现在更多应用程序中。相比由此产生的回报，你的IT投资已经降低了很多，因此软件也就变得更有价值。这是一个互惠互利的局面。

虽然对应用程序的投入减少了，但这并不意味着你愿意降低利润。集成他人代码而生成的应用程序可以卖出不错的价钱。而且，比起其他竞争者推出的产品，它们可以早些面世，从而抢得市场的先机，占据主要市场份额。如同一句老话“早起

的鸟儿有虫吃”，这放在软件开发领域也是真理。如果你的产品能够成为最新最热门的应用，就算借鉴了他人的工作成果也没关系。潜在客户只在乎软件是否可以完成他们需要的功能。他们关心的是这个软件能够为他们做什么，对软件是怎么完成工作的却没有太大兴趣。

善于利用他人代码也会给程序员自身价值增添砝码。一些程序员认为，亲自编写代码能够给他们带来“就业保障”。“因为我能编写出好的代码，我就能永远保住饭碗。”他们这样辩解。问题是，编写好代码需要时间。如果亲自编写应用程序中的每一行代码，反而会显得你工作进度缓慢，效率低下。那些能够迅速有效地裁剪和组合模块的开发人员才真正拥有“就业保障”。如果有这个能力，开发人员往往能在很短的时间内写完很多软件，企业普遍会认为这些员工才是无可替代的人才。

我回想起我认识的一位二流软件工程师，他的编程功底其实很薄弱。但他有一个诀窍，那就是善于将小模块组合在一起。他几乎不怎么亲自编写代码。他只是整天在系统目录和源代码库中寻找那些用得上的子例程代码，并将它们组合在一起拼凑成一个完整的程序。但愿他不会碰上需要亲自编写代码的情况。说也奇怪，不久管理层就发现他是一名优秀的软件工程师，能够按时在预算范围内完成项目。他的大多数同事从来没有意识到，他其实连编写一个基本的排序程序都有困难。不过，凭借着自己善于利用一切资源的能力，他获得了巨大成功。

5.1.2　避免 NIH 综合征

最优秀的组织中也会存在NIH综合征。当一个团队拒绝承认另一个团队开发的应用程序的价值时；当人们宁可自己从头开始编写程序，而不是使用“现成”代码时；当人们压根不愿意使用其他软件，只因为它们是由别人编写的时候，就是NIH综合征在其中作怪。

与大家的普遍看法恰恰相反，NIH综合征并不能开拓人们的视野。查看过别人的工作并夸口说自己可以做得更好，不一定就代表你更有创造性。如果你将现有的应用程序推倒再重新设计，那只是模仿而不是创造。如果能避免NIH综合征，你会开启一扇大门，通往一个令人兴奋的新奇世界。改写现有程序花费的时间越少，你就更多时间用来开发新功能。这就好比在玩“大富翁”游戏，一开局你就拥有了“海滨帝国”和“公园广场”的酒店。这样的话，你便不需要浪费掉一半时间打基础，可以去努力积蓄资本并建设酒店了。

在当前软件产业特别强调标准化的情况下，NIH综合征尤其显得危险。标准驱动着软件供应商逐步走向软件商品化的模式。所有的电子表格产品看起来很相似，所有的文字处理软件也都提供相同的功能，等等。因此，市面上有大量可用的软件能完成日常任务，这造成了供过于求的局面，从而使得软件价格逐步走低，也降低了企业的利润。为了继续留在这个商业竞技场中，每家供应商都需要电子表格、文字处理等软件。但是几乎没有什么供应商能负担得起从零开始编写这些日常软件的开发费用。用行话来讲，那些能够借用软件并对其作出改善或增值的公司才会最成功。

我曾经在一个为窗口系统开发图形用户界面的软件团队中工作过。我们计划去模仿另一款市场上流行的用户界面。因为那款界面在市场上取得了巨大成功，所以我们推断自己的产品肯定也会成为大热门。我们的计划是重写整个用户界面，让它能够更高效地运行。

我们有两个明显劣势。首先，为了编写出一个更高效的解决方案，我们不得不采取一些特别步骤，它们会使得软件不可移植。想要实现我们的目标架构就需要通过“硬编码”方式来开发这个应用程序，这会严重限制潜在市场的规模。其次，从头开始编写一个新的用户界面需要好几个月。在我们忙着编写它的同时，我们模仿的产品团队不会完全闲着。他们也会忙着优化并加入新的功能。等到我们发布自己的版本时，他们的产品至少会比我们先进一代。

幸运的是，盲目的我们并未意识到，在开发自己产品的同时，其他公司的用户界面将会有很大的变化。相反，我们开始担心，如果我们产品的外观和感觉类似于那款我们致力模仿的产品，很有可能会卷入专利侵权诉讼案中。因此，我们去咨询公司一位律师的意见。他提到了一个更有趣的可行方案，令我们眼界大开。

“为什么要去重复其他公司的工作，而不是干脆在我们的产品中使用他们的软件呢？”他问道。我们都倒抽了一口凉气，“自豪感”这个词卡在我们的喉咙，却无法说出口。他的建议真正击中了要害，长久以来我们都精心呵护并守望着自己的NIH综合征。当然，最难就是承认他的想法确实很有道理。

因此，我们开始了解该如何将其他厂商的软件整合到我们的产品中，并加强它的功能。这是份苦差事，尤其得考虑我们之前还想过另起炉灶呢。最后，我们发布了这款以其他软件程序为基础的应用。结果呢？客户对我们在兼容性方面作出的努

力赞不绝口。他们购买了我们的产品，不单单是因为它的价值超出了竞争对手的解决方案，而且它还符合了工业标准。我们有了一个成功产品。

请大家注意“增值”这个词，它是在20世纪90年代及之后的软件领域取得成功的关键。因为计算机硬件早已成为一种商品，所以软件也会继续沿着这条相同的路径前行。各种正在进行的标准化工作基本已经确保了这点。软件的价格会持续下降，因为每一个主要厂商都能提供大同小异的功能。软件公司将有两个选择：要么坐看自己的利润空间被压缩为零，要么通过增值标准化应用程序来维持利润空间。它们必须在保留标准兼容性之余，创造出一些独特的应用，将自己与其他同样具备行业标准的产品区分开来。为了生存，公司必须要接受这个矛盾性的挑战目标，既要满足统一标准，又要保证独特性。想在市场立于不败之地，公司就要增加产品的闪光点，而不是另起炉灶，从零开始。

听起来像极了开源运动中人们互相打气的口号，对吧？你开始上路了。稍后我们会看到Linux供应商和咨询顾问是如何接受增值观念，并将其转变成为一个可行商业模式的基础。

当前商业环境下，NIH综合征信徒在几个方面都备受打击。新型大容量存储技术如DVD就构成了一个巨大威胁。这些用来发布最新电影的闪亮碟片也可以很便宜地存储超过100 GB程序和数据。这种平价的存储介质很有可能永久改变软件业的格局。CD和DVD光盘成了平常事物，存储技术早已发生了不可逆转的飞跃。

最近我参加了一次本地PC展，并发现了一个存储着2 400多个程序的光盘，它的售价仅为5美元。（诚然，这算是目前低端技术的代表例子，但它进一步说明了我前面阐明的道理。）人们很难自圆其说地去重写一份在别处售价不到一分钱的软件，除非你是一位不食人间烟火的开发人员。

也许，NIH综合征最大的威胁来自高速互联网接入技术，比如线缆调制解调器和DSL。这些技术使得任何人都可以免费下载到数以千计的程序。他们甚至不需要到PC展上去寻找代码。点击几下鼠标，一切便唾手可得。

当好的软件几乎能免费提供给大众，NIH综合征就完全没有了市场。完全不借鉴兼容他人成果的软件开发起来代价都过于昂贵。当然，我并不是说人们不需要编写新的代码。大多数新软件都会加强和扩展现有的软件，或实现一个全新的应用。Linux和开源社区很独特，它们能够同时利用这两种优势。

5.1.3 允许他人使用你的代码来发挥软件杠杆效应

软件工程师都喜欢囤积自己的源代码。就好像自己编写了什么可以改变全世界的独特发明或是神奇公式，而且只有自己才知道这个“秘密武器”。他们担心如果公布软件的源代码，那他们便再也无法拥有这颗价值连城的神秘明珠。

首先，软件绝对没有什么神奇公式。任何一个有着合理逻辑思维的人都可以编写出像样的代码。你可能聪明绝顶也可能平庸无奇，但归根结底，所有软件都是一些经过精心计算过的语句，让硬件来执行某些明确定义的动作。看不到源代码的程序员也可以将好程序反汇编出来。反汇编的过程虽然缓慢、繁琐，但还是可以完成的。

那软件的控制权问题怎么办呢？在计算机世界中，人们普遍持有的一个观点是，谁能获得源代码的所有权，谁就能“拥有”该程序。这种说法有一些道理。掌握了源代码的公司能够保留对程序进行修改的权利，或是谁可以得到它的运行许可证，等等。然而，这种控制软件生命周期的行为只能保障短期内公司在该项目开发上的利益，而不是软件本身。它不能阻止模仿者（克隆者）去设法模仿它的特点和功能。大多数值得让一家公司为之投入大笔资金的好创意，也会吸引到该公司的竞争对手。其他厂商迟早会努力赶上这波浪潮，制造出相应的复制品。最成功的软件是那种能够尽可能出现在更多计算机中的软件。以专有软件方式运作的公司会发现这种做法并不利于它们的长期发展。

Unix的成功在很大程度上要归功于一个事实，那就是它的开发人员认为执意保留着源代码的控制权并无太大必要。大多数人都觉得这种做法没有任何意义。早期他们把Unix视作是一个有趣古怪的玩意，很适合实验室和大学环境，但仅此而已。没有人（除了它的开发者）会认为这是一款真正的操作系统。因此，人们只需要花费少许代价就能够得到它的源代码。

飞涨的开发成本迫使硬件供应商逐步减少在软件方面的投入，尽管这些投入可以让它们的硬件平台更具市场竞争力。Unix这款操作系统的不利因素反而很快被人们视作是一个优点。它也因此得到了蓬勃发展。每当人们想为新的计算机节省操作系统的开发费用时，他们就会想到Unix。即使在今天，许多人仍然认为在可选的操作系统里，它是最经济实惠的。

今天，Linux的低成本使它成了很多软件公司的首选平台。基于Linux系统开发者提供的内核、编程接口和应用程序，这些公司充分利用了软件杠杆效应。在节省了编写操作系统和应用程序集的成本之后，它们可以集中精力改善自己的应用程序，提供优质、增值的定制解决方案。因此，比起世界上其他那些需要投入资金开发操作系统后才能进行应用程序开发的软件公司，这些公司处于更有利的位置。经常采取这种运作方式的公司可以出售自己的专业知识，而不是软件本身。软件公司就变成了提供咨询和定制服务的机构，而不是软件的开发商。

5.1.4　将一切自动化

一个能最大程度利用软件杠杆优势的有效方法就是让机器加倍努力工作。今天，如果你还在手工完成任何计算机能做的事情，就纯粹是在浪费时间。可即使是在现代工程实验室中，还是有很多有经验的人士仍然依赖着原始的手工方法来完成日常工作，这可真是奇怪。其实他们更应该了解其中的道理，可就是积习难改。如果你也是自己在拼命努力工作，而计算机却呆在那里无所事事的话，下面列举的一些小窍门也许会对你有所帮助。

- 你是否经常使用纸本打印的文件？一旦数据或文字打印在纸上，它的管理就变成了手工操作。这种做法非常低效，更何况还浪费纸张。传统风格的企业管理者往往喜欢这么做。
- 你是否还在人工处理数据排序或是计算行数和对象个数？多数操作系统环境，特别是Linux提供了一些工具来执行这些任务，它的处理速度远远超过人工排序的速度。
- 你如何查找系统上的文件？是通过浏览一个接一个目录来定位吗？还是创建一个自己的文件列表，用编辑器或浏览工具对其进行扫描呢？在这些情况下，`find`、`grep`和`locate`等Linux命令可以组合成一个强大的工具。
- 当试图在一个文件中寻找特定项目的时候，你是靠自己肉眼来扫描文件定位查找目标？还是会采用浏览器或是编辑器的“搜索”命令，让系统为你做扫描？
- 如果命令解释器提供历史机制，你是否会使用这个功能？历史机制可以通过快捷方式来调用前面使用过的命令。Linux下的shell如`csh`和`bash`在这个方面表现得都很出色。

- 如果有一个多窗口的系统，你会一次只打开一个窗口吗？其实，同时打开两个或多个窗口会让你大有裨益。你可以将一个窗口作为工作空间（进行编辑，编译等动作），其他窗口则变成测试空间。更妙的是，一些Linux窗口管理器还提供了“多桌面”功能。它能够以组为单位管理多个窗口。由此往往能大幅度提高工作效率。
- 你使用剪切和粘贴工具的频率高不高？如果需要经常手动输入长长的字符串，那你很可能没有充分利用好这个功能。我认识的一个人就经常开着一个窗口，里面包含着他惯常使用的字符串。在需要的时候，他会从窗口复制需要的字符串，然后粘贴在其他的窗口中，这可省掉了很多打字的工夫。
- 你使用的命令解释器会提供自动补全命令和（或）文件名的功能吗？你是否会使用这些功能以加快输入，免去那些不必要的多余按键动作？

通过加强自动化工作来利用软件的杠杆效应能产生巨大的生产力。我曾经在一个地方工作过，其中很大一部分工作涉及从不同来源中筛选收集少量的信息。我们做过一项研究，观察到每一个人每周花费在查找所需数据的时间多达20个小时。这个数字还不包括在定位到信息之后，花在浏览和核实上的时间。于是，我们编写了一个简单的工具，对多个来源的信息做了索引，以备将来快速检索。这么做的回报是，使用该工具之后人们每周只要花3个小时，就能定位到以前需要花上超过15个小时才能找到的同等规模的数据。人们的产能激增，每个人都兴奋不已。曾经繁琐复杂的研究工作现在一点都不麻烦，这种高度互动的一系列查询动作都由机器来完成。它节省了时间，让我们可以腾出手去解决其他更难的、机器无法处理的问题。

每次当你想自动化完成某个任务的时候，你都能体会到我的姑姑发挥杠杆效应，让其他人来为她出售特百惠产品的喜悦感。每个命令处理和调用程序都会让计算机不知疲惫地去完成一项任务。你不需要说服其他人来完成你的工作，只需要指挥一台配置良好的机器按照指令去执行预设好的任务就可以了。机器运行的速度越快，杠杆效应也就越强。等到明年的新款机器出现时，你所拥有的杠杆作用将会产生更大的影响。当然，机器从来都不知道疲惫，也不会想要分得更多利润。

本章的第一部分，我们从总体角度及该如何将杠杆效应应用在软件上的角度讨论了一些使用杠杆的原则。我们已经看到了，成为一名“软件清道夫”并时刻留意是否能够利用他人工作的重要性。在讨论完恼人的NIH综合征之后，我们强调了分

享资源的价值。最后，我们还讨论了一个显而易见但常被人们忽视的事实，那就是使用计算机将日常工作尽可能地自动化，这样可以更充分地利用好时间。

现在，我们已经奠定了一个基础，接下来该采用其他Unix元素（也就是shell脚本）来扩展这个基础。在软件中运用杠杆效应的方式中，shell脚本发挥着非常有趣的作用。它们不单可以让新手用户进入Unix这个令人难以置信且潜力无穷的世界，对专家级用户也一样。有经验的Unix程序员都在虔诚使用shell脚本，你也应该这么做。

shell脚本与其他一些命令解释器和控制机制有着相似之处，比如MS-DOS下的批处理文件和OpenVMS的DCL命令。不过，shell与这些机制的不同之处在于，Unix的shell脚本运行在一个非常适合间接执行命令的环境里。为了突显其重要性，我们在Unix哲学中包含了一条专门针对shell脚本的准则。

5.2　准则7：使用shell脚本来提高杠杆效应和可移植性

如果你想充分利用软件的杠杆效应，就需要学习如何有效使用shell脚本。在这里，我们不会向你展示它的语法细节。市面上早就有很多关于这一主题的书籍。大部分都会教你具体使用方法。这里我们将集中讨论为什么应该使用shell脚本。

开始讨论之前，我必须提醒你，许多Linux和Unix的内核程序员对shell脚本有点不屑一顾。他们认为，编写shell脚本并不是一种具有大丈夫气概的做法。有些人甚至将“有经验的shell程序员”等同于“轻量级的Linux程序员”。我的猜测是，他们纯粹是“羡慕嫉妒恨”，编写shell脚本可不会让人们感到头痛欲裂，也可能是因为内核里不能使用shell脚本。或许哪一天某些人将编写出一种内核，允许shell脚本在内核空间中使用。这样，那些不喜欢shell脚本的程序员就能更充分地感受到软件杠杆效应带来的好处。

在讨论shell脚本的时候，你也许会觉得我是一个C语言的反对者，也就是说，你觉得我会建议人们应该永不采用像C这样的可移植语言来编写程序。如果你这样认为，那你就完全曲解我的意思。有些情况下，采用C编写程序要比使用shell更加合情合理。但是，这种情况发生的概率往往比人们猜想的要小得多。

同样，由于最近这些日子以来，围绕在面向对象语言和工具上的大肆炒作，人们很容易掉进陷阱，放弃shell而采用某些时髦语言（如Java）来编写所有的软件。

虽然在许多方面Java的确是一种优秀的编程语言，尤其在代码重用上，但它只是一个编译性语言（也就是说，在Java运行引擎能够解释它之前，你必须编译Java源代码而得到运行程序）。就算不能彻底扭转你的看法，本节也会让你认识到，除Java之外，shell也是一种不错的选择。

5.2.1 shell 脚本可以带来无与伦比的杠杆效应

shell脚本由一个或多个语句组成，通过调用本地程序、解释程序和其他shell脚本来执行任务。它们将每条命令都加载到内存执行，并且间接调用这些程序。顶级shell程序根据语句的种类，可以选择是否要等待单个命令完成它的执行任务。它调用的命令是早前就编译好的程序，大部分都不是由你自己编写的，这些命令的C语言源代码行数可能多达一百、一千，甚至十万。其他人花费时间编写并调试通过了这些程序。你编写的shell脚本就成为受益者，它最大程度地利用了这些代码。因此，虽然你只付出了相对较少的努力，却受益于高达一百万行或是更多行的代码宝库。这就是软件杠杆效应的影响力。

特百惠塑料容器多层次市场营销中的关键就是让别人来帮你做大量工作。你要极力营造一种他人播种、你来收获部分成果的局面。shell脚本就提供了这样的机会。它能够集成他人的努力成果以满足你自己的目标。你不需要重复编写shell脚本中使用的大部分代码，因为别人早已帮你完成了这个工作。

让我们来看一个例子。假设你需要一个能在单行输出结果里列出当前系统上所有用户名的命令。为了让这个问题更有意思，无论用户在系统上打开了多少会话窗口，我们只显示一次用户名，并用逗号分隔每个用户名。下面就是这条命令，它是针对流行的Linux命令解释器bash编写而成的shell脚本：

```
echo `who | awk '{print $1}' | sort | uniq` | sed 's/ /, /g'
```

虽然这个shell脚本只有一行，它却调用了6种不同的可执行命令：echo、who、awk、sort、uniq和sed。这些命令几乎同时运行，成为一系列平行的执行例程。除了最先启动的who命令，其余命令都从前一个命令那里接收数据并将输出发送到下一个命令。其中的“|”字符是管道，它们管理着数据的传输方向。序列中的最后一个命令sed将它的输出结果发送到用户终端。

每个命令都和其他命令协同工作来产生最终输出结果。who命令将系统上的用户名单以多列的形式显示出来。这个输出结果通过管道机制传递给awk命令。awk命令只保留了第一列包含用户名的输出结果，并舍弃who命令产生的其他数据。然后，这份用户名列表被发送给sort命令，它会将它们按字母进行排序。uniq命令则舍弃掉那些因为用户在同一时间登录了多个会话窗口而重复出现的结果。

现在我们有了一个排好序并采用Linux的行尾符（或是换行字符newline）分隔的用户名列表。此列表通过反引用（back-quoting）机制发送给echo命令，也就是将前个命令的输出结果当作echo的命令行参数。在bash的shell语义中，它会采用单一空格来取代所有换行符。最后，这个由空格分隔的用户名字符串会被发送到sed命令，其中的空格被转换为逗号。

如果从来没有接触过Linux系统，你可能会觉得这真是一行神奇的代码，但这只是典型Linux风格命令行的执行过程。在shell脚本中，单个命令行里调用多个命令的现象一点儿也不奇怪。

在这个过程中，到底执行了多少行代码呢？ shell脚本的编写者只花了不到一分钟就写完了这个脚本。该脚本调用的命令其实是由前人编写好的。在今天的Linux版本中，此脚本调用的6个命令包含的源代码行数如表5-1所示。

表5-1　6个命令包含的源代码行数

命 令 名	行　数
echo	177
who	755
awk	3 412
sort	2 614
uniq	302
sed	2 093
合计	9 353

由此可见，仅这一行shell脚本就执行了总计9535行源代码！尽管这个数字不算非常可观，但这个行数也足以证明我们的观点。在这里能量放大的比率是9353：1。再次强调一下，我们应该充分利用软件的杠杆效应。

这是关于shell脚本的一个简单例子。今天，一些shell脚本的篇幅长达几十页，包含着数百行命令。把每个可执行命令背后的C语言代码也算进来的话，你可以算

出加在一起的实际数目是多少。shell脚本产生的杠杆效应令人叹为观止。正如我们将在后面看到的，这种现象甚至给爱因斯坦都留下了深刻的印象。

5.2.2 shell 脚本还可以充分发挥时间的杠杆效应

shell脚本有一个天生的优势就是它们是一种解释型的语言，不需要编译。在标准的C语言开发环境中，构建程序的处理顺序是这样的：

思考→编辑→编译→测试

在shell脚本开发人员的环境中，这个步骤更短：

思考→编辑→测试

shell脚本的开发省掉了编译这个步骤。由于今天的编译器都是经过高度优化的，这看起来可能并不是什么了不起的成就。在快速的RISC（精简指令集）处理器环境下，一眨眼的功夫这些编译器就能将源代码编译成二进制执行文件。不过今天的应用程序几乎很少是独立存在的单个文件。它们往往散落在大型项目构建环境中，让情况变得尤为复杂。或者因为整合的程度越来越高，过去在一个快速机器上只需要几秒就能完成的编译过程，现在可能需要一分钟，甚至更久。规模较大的程序可能需要几分钟或是更长时间。完整的操作系统及其相关的命令可能需要数小时编译时间。

省略了编译步骤后，脚本编写者能够继续专注于开发工作。他们并不需要去喝杯咖啡或收取邮件来等待命令的编译完成。人们可以直接从编辑过渡到“测试”过程，思路不会因为编译过程而被打断。这大大加快了软件开发的进程。

还有一个需要考虑的关键因素就是执行时间与编译时间的比较。许多较小的应用程序（请记住，我们这里讨论的是Linux或Unix系统）只需几秒就能完成任务。如果编译时间过长，那么使用脚本就无可厚非。另一方面，如果你必须耗费数个小时来运行脚本，而一个编译好的程序在几分钟内就可以执行相同任务的话，那么你最好编写一个C程序来完成这项任务。你只需要确保自己已经仔细考虑过是否有另外可行的节省时间的方式，比如采用组合了各个模块的脚本。

相比shell脚本作者，C程序员还是有他们的优势，即拥有一个能够用于调试的强大工具集。开发人员早已为C语言创建了一大堆数量可观的诊断软件，而脚本开

发者的诊断工具却极为有限。迄今为止，还没有功能齐全的shell脚本调试器问世。shell脚本作者仍然必须依靠原始机制（如sh -x）来显示该脚本执行的命令。能够就没有能够设置断点的便利工具。变量的检查是一个繁琐的过程。人们可以争辩说，shell编程如此方便，不需要有全面的调试工具。我表示怀疑，大多数shell脚本编写者恐怕也不会同意这个看法。

如微软Visual Studio或Borland公司的JBuilder这样的IDE呢？难道它们不是优越的调试和编辑工具吗？答案是肯定的。可它们太善长对程序员隐藏底层开发技术的复杂性。有时候这是一件好事。但人们经常会抱怨出错的时候还是要去查阅一下掩盖着的底层状况，看看IDE究竟做了些什么。在试图找出幕后到底出了什么错的时候，让编程语言易于使用的漂亮外层却成为了开发人员的克星。

另一方面，shell脚本却很直白。它们所做的一切就摆在你的面前。不会有什么细节隐藏在花哨图形用户界面的下拉菜单之下。具有讽刺意味的是，比起所谓的“直观”（visual）产品，shell脚本反而更直观。

5.2.3　shell 脚本的可移植性比 C 程序更高

一个确保能利用软件杠杆效应的方法就是使程序具有可移植性。之前我们就了解到，与他人分享自己的软件有很重要的意义。那些能够毫不费力地从一个平台转移到另一个平台的程序更能为大家所用。你的软件用户数量越多，它产生的杠杆效应就越大。

在Linux的环境下，shell脚本一般意味着最高级别的可移植性。大多数能在某个Linux系统上工作的脚本，很有可能稍加或干脆不加任何修改就能应用于另外一个系统。由于它们都是解释型脚本语言，因此使用的时候也就没有编译或是转换的必要。当然，你也可以故意将一个shell脚本设计为不可移植的，但这种情况非常罕见，通常情况下也不鼓励这样做。

shell脚本往往也没有与C源代码紧密相连的“所有权”问题，人们很少会去保护它们的所有权。由于这些脚本对每个人都清晰可见，因此大家都不认为自己的公司有责任看管其中内容。尽管如此，还是有人采取了谨慎的措施。美国和其他国家都提供了相应版权法来保护shell脚本的版权。如果你想要确定shell脚本的版权归属，应该交由律师处理。

5.2.4 抵制采用 C 语言来重写 shell 脚本的愿望

在关于可移植性的那一章中，我就建议过大家不要采用C语言来重新编写shell脚本，因为机器运行速度会越来越快。由于shell脚本通常都是高度可移植的，因此将它们转移到新的机器一般都不费什么力气。你只需将它们复制到新的机器，它们就能运行。不麻烦，也没有值得大惊小怪的事情。

然而安于现状并不是一个程序员的传统美德。如果程序员找得出空闲时间来对shell脚本进行完善的话，我可以打赌，他们肯定愿意增加一些新功能，或试图通过优化脚本来使其运行得更快，要么就通过采用C语言来重新编写部分或是大部分代码来改善其性能。你能猜到程序员最青睐哪个做法吗？

程序员这种采用C语言来重新编写shell脚本的强烈愿望，源于他们认为C程序的运行速度比shell脚本要快的信念。建立一个整洁有序的世界，一切都被高度优化过而且非常高效，这种想法战胜了一切，成为程序员的崇高目标。这其实是涉及自尊心的问题。程序员觉得应该用C语言来编写这些程序，而且它们会运行得比shell脚本快。无论出于何种原因，他们没能这样做，然后内心就会充满负疚感。如果你问他为什么选择编写shell脚本程序，他只会咕哝一句“我只有这么点儿时间”。他还会进一步表态，并承诺如果时间允许的话，将会用C语言改写这个程序。

是时候忘记这回事儿了。我很怀疑他是否有这个机会。任何劳动都与回报成正比，拿着相应薪水的程序员太忙了，没空回头去重写一个本来就运行良好、足以满足用户需要的shell脚本。人生苦短，别浪费时间在这上面。

此外，C程序比shell脚本跑得快的想法还有待审议。首先，shell脚本调用C程序来完成其任务。一旦将C程序装载到内存中运行，纯C程序与那些从脚本中调用的C程序相比，并没有太大的性能优势。今天，大多数Linux系统已经对命令的执行例程做过优化，相对而言，加载程序运行所需要的时间只占完成整个任务的时间的很小一部分。

如果真的想让shell脚本运行得更快，那么你就必须找到不同的解决方法。很多时候用户和程序员容易产生思维定势，辩白那就是他们一直以来采用的处理方法，不能随意更改。但是，你需要锻炼自己，不要拘泥于这些一目了然的方法和技术，去寻找使用现有资源的创新方式吧。

作为例子，让我们来看看几年前我碰到过的情况。在曾经的工作环境里，我每天都要收到50~100封电子邮件，每星期大约300封。虽然我读完一些邮件后我可以将它们删除，不过我还是要把其他一些邮件保存备份。不久，我就积攒了2000多封电子邮件，它们分布在我Unix系统下100多个目录中。想要找到某封邮件变得困难重重。

简单的解决办法就是使用Unix的grep命令在邮件内容中定位特定文本字符串，从而找到所需邮件。这个方法的问题在于它非常耗时，虽然我已经采用了像grep这么快的程序，但我还是需要更好的解决方案。

几次不成功的尝试之后，我想出了的主意就是为所有邮件做好索引。我编写了一个shell脚本，使用所有可能的文本字符串对目录下的每一个文件运用grep命令。实际上，我创建的索引正是对文件进行“预grep”处理的结果。当我想找到一封邮件时，我可以在索引中查找这封邮件包含的文本字符串，比如作者或是主题。该索引便会返回一个指针，指向这封邮件或是包含了该字符串的所有邮件。比起每次运行grep命令来查找，这种做法更为行之有效，而它只使用了一个shell脚本。

这个方法效果非常好，于是我便将它分享给一位同事，他正在实施一个更大规模的类似任务。短短几个月，他便使用同样的技巧对我们系统上的海量文件构建索引。他还改进了这些shell脚本，最终在几秒内它就可以定位到几百兆字节文本中的一个字符串，这还是在我们最慢的机器上运行。这个应用程序效果如此显著，以至于大多数人都不相信它是一个纯粹用shell编写的脚本。

虽然grep的运行速度远远超过shell脚本中调用的其他许多程序，不过采用不同方法来运行shell脚本的效果，可能比单个grep命令的效果要好。我们只需要从全新的角度来看待这个问题。

在这一章中，我们探索了利用软件杠杆效应的价值所在。我们已经看到了，在软件中应用杠杆优势是一个多么强大的想法。如同任何形式的“复利”一样，通过软件的杠杆效应，你可以只付出少量的努力，却得到广泛影响。每个小程序都是一粒种子，精心播种之后便能成长为参天大树。

shell脚本仍然是提高软件杠杆效应的最佳选择。让你得以利用他人的工作，并发挥出你自己的优势。即使你从来都没有写过什么排序例程，也可以随心所欲地使用由专家编写的排序程序。这种做法使每个人都成为赢家，甚至那些不太高明的程序员也不例外。

Unix哲学的优势之一就是它很重视数量众多的小命令。shell脚本是一种将它们统一在一起成为一个强大整体的媒介。从而使初级程序员都能够轻松完成复杂的任务。使用shell脚本，你可以站在巨人的肩膀上，这些巨人其实也站在其他巨人的肩膀上，循环反复。这就是软件的杠杆效应。

爱因斯坦曾经说过："在我的生命中，我只见证过两个奇迹，核聚变和复利。"在他知晓的所有美妙理论中，这两个现象显然给他留下了最深刻的印象。他深深地懂得小事物在反复放大之后，可以聚集起到不可思议的能量。也只有像他这样具备敏锐思维的人才能意识到这个简单想法的威力。

另一方面，也许他的妻子曾经卖过特百惠。

第6章

交互式程序的高风险

大众汽车公司是对的。确实，小即是美。真正的力量不在于有多强大，而是小而有能力。你只需看看今天有多少小型汽车行驶在美国公路上，就会意识到几百万人都认同这个结论。而且，大众汽车公司在复古型甲壳虫汽车上市时再次推行了这个理念。小就是潮流，它是永远的潮流。

同样，人们对小型化概念的喜爱不局限于汽车。人们发现，相比规模更大的同类产品，小东西有着巨大优势。平装书销量量超过精装版由来已久，部分原因是因为它们比较便宜，另一个原因则是因为它们更容易随身携带。手表之所以取代怀表也是因为它们的体积更小，便于携带。技术的进步使得今天这些微型电子组件的能力超过了往日那些大型组件。掌上电视机、掌上电脑、手持遥控器的销售额呈井喷态势。如今，就算是核武器的尺寸也要远远小于第二次世界大战期间投放在日本的那两枚核弹，而它们的破坏力却大得多。

世界上的电子产品在日趋小型化，我们要把这种现象归功于卓越的科学技术。将电脑主机缩小到人的手掌那么大，需要非常先进的技术。如果没有高密度微处理器提供的小型化技术，今天的许多产品都将过于庞大而无法良好使用。

不过，用户还是要花费一些心思才能用好这些高科技成果。对于那些越变越小、越来越先进的设备，用户需要学习更多相关知识才会知道如何使用。

家用微波炉的演变是一个代表性例子。早期的微波炉只有一个启动按钮和一个简单的计时器旋钮。然后，随着计算机芯片的价格持续下跌，生产可编程的微波炉成为一种潮流。微波炉变得相当智能。当然，也只有更聪明的用户才能利用上它们这些先进的功能。

我们可以从另一个角度来看待这个问题。在孩子们学写字的时候，他们最早使用的并不是那种极细的记号笔，而是用的大号蜡笔。为什么呢？因为孩子们能更好地掌握蜡笔的抓握画写等精细动作。用极细记号笔写字需要高超的手部技巧。孩子们只有在先拿起蜡笔练习数年之后，才能掌握这个技能。

从这些例子我们可以得出两个结论。首先，小东西与人的交互性要差一些。虽然微技术使得设备变得更小，但人却没有缩小。一旦物体缩小到一定程度，人们就无法掌控它们。人类必须依靠工具来提高正常的身体感官技能。例如，钟表匠无法用肉眼审视精密瑞士手表内的小部件。他必须借助放大镜才能看清楚里面的零件。同样，在今天的半导体制造工厂，人们必须通过显微镜来查找集成电路的缺陷，因为每平方英寸芯片上都容纳着数百万个晶体管。

人的感官功能非常具有局限性。人们无法听清太大或太小的声音。自然界存在着一些人类肉眼无法察觉到的低频率光波。我们可以凭借嗅觉来区分香水的品种，或是臭鼬散发出的恶臭，但我们却无法分辨两种不同臭鼬的气味。

技术驱动下电子设备变得越来越小，最终它们会小到我们再也感觉不到程度。于是，人们需要通过特殊接口来对它们进行操作。随着计算机技术日益成熟，存在和被感知之间的差距将逐渐扩大。执行既定任务的软件和它的用户界面之间也会存在着一个不断扩大的空白。

我们对小型事物作出的第二个结论就是，在小型事物与人的交互性变差的同时，它们之间的交互性却会大大增强。小尺寸赋予了它们极大的灵活性。在很多情况下，它们能够轻而易举地整合在一起。

让我们参考一下这个例子：下一次你看到搬家用的货车时，可以留意一下工人们是如何装载客户物品的。如果他们打算把客户的汽车也放到卡车上，那他们最先安置的物品就会是它。然后，他们再决定次大的物品应该放在哪里。由此类推，接下来是中等大小的物品，然后才是最小的物件。人们理所当然地会这么装载货物。这个例子向我们生动地说明了，小东西能以无数方法组合在一起去完成一项任务。在小物件拥有更多装载灵活性的同时，那些大的物件却没有这么多选择。

那如果你只拥有小物件，会发生什么状况？你会获得最大的灵活性。可这种灵活性要付出代价。你拥有的小物件越多，操控它们就越难。管理这些小物件便成了一个严重问题。

同样，在计算机软件领域里，拥有许多小程序和模块会获得最强的环境适应能力。不幸的是，随着这些模块逐渐变小，它们在与用户交互时就会出现更多问题。模块数量越多，操控起来就会越复杂。

这让软件设计人员陷入进退两难的窘境。程序员想为应用程序获取最大的灵活性，因此便构造了一个小模块的集合。然而，他还受制于软件要易于使用的需求。毕竟，在处理过多小模块的时候，人们会感觉困难重重。

Unix系统采用了一种不同的方法来解决这个问题。大多数其他系统试图采用一种被称为强制性的用户界面（CUI）来架设一座用户与模块之间的桥梁，从而弥补这种日益扩大的差距。Unix开发人员也认识到这个问题，但他们并没有将一大堆“意大利面条”代码（也就是凌乱无章的代码）来把用户和底层模块联系在一起，而是采用小程序块或层来应对这个差距。

我们刚刚说过Unix和其他操作系统在这一点上产生了重大分歧，你可能会怀疑这种做法是否合理。Unix哲学这下一条准则可以打消你的疑虑。

6.1　准则8：避免强制性的用户界面

在开始讨论为什么要避免强制性的用户界面（以下使用CUI表示）之前，我们首先需要定义一下到底什么是CUI。CUI是一种与应用程序进行交互的模式，它位于系统最高级命令解释器之上。一旦你在命令解释器中调用了一个应用程序，那么直到应用程序退出之前，你都无法再与命令解释器进行交互。实际上的效果就是，你完全被这个应用程序的用户界面牵扯住，直到你退出之后才能重获自由。

让我们通过一个例子来说明这种情况。假设你有两个程序，一个程序会列出电子邮箱中的内容，另外一个是对文件中的文本字符串进行搜索。如果邮件程序和搜索程序都使用了CUI，那你与它们的交互过程看起来可能如表6-1所示。

表6-1　CUI交互过程

`$ mail`	从命令行调用mail程序
`MAIL>dir`	显示邮箱中的内容
`MAIL>exit`	退出mail程序，返回到命令解释器这一层
`$ search`	调用search程序

（续）

SEARCH>find jack *.txt	执行search
SEARCH>exit	退出search程序
	:
	::
$	回到了命令解释器这一层

请注意，整个操作过程中命令层次的变化。首先，调用mail命令会将你带入它自己的命令解释器中。迫使你只能与mail命令解析器进行交互。这个解析器的命令集和之前调用它的那个命令解释器的命令集完全不同。其次，为了执行search命令，你必须先输入exit以退出mail命令，然后，再从主命令解释器去调用search命令。一旦你启动了search应用程序，你又必须与它的命令解释器进行交互。其行为不但与主命令解释器不同，与mail命令的解释器也不一样。唯一的相似点就是，它们都要求你输入一个exit来作为结束。

你可以看到，这种方法有着明显缺陷。你必须掌握三种不同命令解释器的使用方法，每一个都有着自己的交互语言。小范围内，这听起来可能还不是太难，但在一个拥有着数百个应用程序的系统上，这就成为了一项“不可能完成的任务”。此外，在执行一个命令的时候，直到退出该命令之前你都无法去做别的事情。比如说，你正在回复一封邮件，而且需要从另一个文件中包含一些文本，但你却忘了到底是哪个文件。那么，你就必须先退出mail命令，执行search命令，然后再返回到mail命令。那时候，你可能都忘记邮件上下文的内容了。

谈了这么多CUI的缺点，但我们还是不清楚为什么Unix信徒非要避开CUI不可。不过，在探讨这些之前，让我们看看Unix风格的非CUI界面，请留意它与前述CUI的差异（参见表6-2）。

表6-2 Unix风格的界面

命 令	描 述
sh>scan	scan命令可以列出邮件文件夹的内容
	:
	::
sh>grep jack *.txt	grep命令在所有以.txt为后缀的文件中查找"jack"字符串

请注意，Unix用户都是在shell提示符下或是主命令解释器这一级来调用所有命令。命令完成其任务后，控制权就返回到shell提示符下。人们不需要特意键入“exit”来退出每一个命令。用户只需要学习一种语言，即Unix的命令解释程序shell。

犬儒主义者[①]可能会指出，用户仍然要学习各个命令的调用顺序或是参数。没错。但是在CUI下，用户首先必须想想他要调用的命令，然后从该级CUI中又该调用哪个子命令。因此，在CUI中完成同样的效果需要花费你两倍多的精力。这并不奇怪，而且，拥有这种CUI的系统往往还得提供高度完善的帮助系统来对用户进行指导。另一方面，大多数Unix用户在没有复杂帮助系统情况下仍然可以自如地运行命令。当用户输入了不正确的参数，Unix命令通常会返回一份简单的消息列表，其中列举着各种所需参数和它们的简单用法。

到目前为止，我们已经定义了什么是CUI，而且讨论了一些它们的明显缺陷。然而，Unix用户对其避之唯恐不及的真正原因更为深刻。那就是他们必须采用一种使命令能够互相交流的方式。在Unix环境中，命令都不是孤立存在的，它们会时不时进行交互。CUI干扰了多个命令进行交互的能力，而让多个命令得以互动是Unix的一个关键概念。

6.1.1　CUI 假定用户是人类

CUI的创建者在设计界面的时候基于一个假设前提：会有人坐在键盘前面去操作机器。他们期望这个人对应用程序的提示键入一些回应。然后，应用程序就可以执行计算过程或完成各项任务。

但问题是，就算是动作最麻利的人也比一般电脑的反应速度要慢。计算机可以在电光火石之间完成它的操作任务，它不知疲倦，不需要休息。如前所述，人的身体机理功能有很大的局限性。举例来说，即便速度最快的打字员每分钟最多也只能键入80个单词。大多数CUI总是需要用户对它的提示作出迅速反应。这种情况下，即使是速度最快的超级计算机，最后的处理效率也只与那些最慢的个人电脑相当，因为几乎所有的PC都能在瞬间捕捉到用户输入的文本。只要系统运行受到人类局限性的制约，它就不能发挥最大潜力的功效。

我第一次意识到这种现象，是在我面对一台高速工作站上的窗口系统时。早在个人电脑出现之前的日子里，大多数人都习惯用终端来检视文本文件。终端通常允许你按下组合键<^S>/<^Q>或“HOLD SCREEN”键来停止或开始输出。在调制解

① 犬儒主义学派是古希腊四大学派之一。奉行这一主义的哲学家或思想家的举止言谈、行为方式甚至生活态度与狗的某些特征很相似，他们旁若无人、放浪形骸、不知廉耻，却忠诚可靠、感觉灵敏、敌我分明、敢咬敢斗。于是人们就称这些人为“犬儒”，意思是“像狗一样的人”。——译者注

调器的速度为9600位/秒或更低的时候，大多数人完全可以毫不费力地控制文本的滚动速度。然而，今天窗口系统显示文字的速度，已经不受到通信速度之类人为限制的影响了。用户基本只能受CPU及其I/O功能的摆布。可以这么说，在任由这些设备自由发挥的时候，电脑显示文本的速度大大地超过人手按下<^S>键所能控制的速度。未来，随着计算机处理器速度普遍达到几G赫兹，以及高速缓存的容量越变越大，这样的状况将会进一步恶化。

由于人类使用计算机的限制，因此任何需要等待用户输入的系统，其处理速度就与人类操作速度相当。换言之，这并不是很快。

典型Unix命令会努力执行自己的任务，尽量做到完全不需要人工干预。大多数程序也只是在它们即将执行一些不可逆操作时才会提示用户，例如通过删除文件来“修复”一个文件系统。因此，Unix命令始终以最高速度运行。这也正是为什么采用“舍高效率而取可移植性“设计理念的系统还会表现良好的原因之一。那是因为，它意识到从系统性能的角度来看，最薄弱的环节就在于许多人机交互的动作，而不是机器本身。

6.1.2 CUI 命令解析器的规模庞大且难以编写

命令解析器读取用户输入人并将其转换成应用软件能够理解的格式。它必须正确读取所有用户键人的内容，包括可想象的和不可想象的。这会导致命令解析器的规模无限膨大。有时候，花在编写命令解释器上的工夫比花在应用程序主体任务上的精力还更多。

考虑一下这个例子。假设有一个硬盘格式化程序。由于数据丢失的潜在危险性很高，因此你可能会认为，询问用户是否真要清空磁盘上的所有数据是一种用户友好的做法，例如：

```
FORMAT V1.0 Rev.A
About to format drive C:
Formatting will destroy all files on the disk!
Begin format?<y|N>
```

回答这样的提示，用户的潜在回应会有很多种，其数量是相当惊人的。首先，如果用户希望继续格式化，他可以键人Y、y、Yes、YES或是各种组合。同样，如

果他确信并不想继续，也许会键入N、n、no或NO。当然，对这些回应的解析暂时还相当容易。

在用户不太确定他想做的事情时，事情就开始变得复杂了。新手用户可能会键入“help”并希望能够获得此格式化命令的大概信息。有经验的用户会输入“?”来获取格式化选项列表。还有一些用户也许会试图通过输入一个或多个中断字符来退出该程序，从而完全终止这个格式化命令。其中一些动作很可能会导致格式化应用程序仓皇退出，甚至终止用户的该次登录会话。

为了应对这些可能，系统需要一个大规模且具有高度精密性的命令解析器，按Chris Crawford的说法就是“在坚硬岩石中开辟出的一条隧道”（参考第9章）。你可以想象如果这个应用程序需要提供多种提示信息，它的规模会有多大。这个功能的代码量将占到总应用程序的很大一部分。

Unix程序员对付用户界面的方法就是——避开它们（比如说，典型Unix应用程序是没有命令解析器的）。Unix命令希望在调用的时候，通过命令行来输入运行参数。这便消除了前面我们提到的多种可能性，特别是不那么优雅的几种处理方式。对于这些有许多命令行选项的命令（它在一开始会给出警示），Unix提供了标准库例程来清理无效的用户输入。这种做法颇有成效，它使得应用程序的规模变小了。

另一种方法就是，使用图形用户界面的对话框。这种做法对于FORMAT这样有着潜在灾难性影响的命令特别有用。人们都不会愿意看到因为自己对字符的选择不当就把硬盘清空了。于是，我们可以使用图形用户界面来放慢用户的节奏，让他们有充足时间思考自己到底要做什么。

尽管图形用户界面可以成功地引导用户穿过这条“坚硬岩石中开辟出的隧道”，但是，它们并不能让你很好地连接这些隧道。钻透这些坚硬的岩石是要大费周章的，尤其是那些巨大的石头。稍后，我们将详细阐述这个问题。

6.1.3　CUI偏好“大即是美”的做法

许多CUI使用菜单来限制用户的选择。理论上这听起来还不错。但出于一些莫名的原因，CUI设计师们很少会满足于选项寥寥无几的菜单，比如说只有5个。他们往往会添加一些能够调用“子菜单”的选项，以扩大选择多样性。就好像他们在

想："嘿，我不辞辛苦地设计了这个菜单系统，也许你们能够利用它做点什么。"最后，这种"功能控心态"压倒了"简洁性"。

当然，计算机行业的营销部门也负有一些责任。他们不断地对软件设计师们施加压力，喊着"功能，功能，功能!"，强烈要求扩充菜单的选择项，而丝毫不理会这些新选项是否真的有用。从销售人员的角度来看，应用程序有好的性能还不够，它还必须要有更漂亮的外观。他们会觉得，如果一个有着5个菜单选项的程序的销售状况颇为理想，那有着10个菜单项的程序就会卖得更好。他们才不关心这些不必要的复杂性可能会适得其反，最终疏离目标市场用户。

避免CUI和它们这种"大即是美"的做法也有一些技术层面上的考量。在CUI变复杂的同时，需要的系统资源会越来越多。对内存的要求也不断增大。人们还需要购买更多的磁盘空间。网络和I/O带宽也存在类似问题。因此，电脑硬件厂商们当然都热爱CUI。

6.1.4 拥有 CUI 的程序难以与其他项目相结合

Unix的一大优势就是，它的程序彼此能够有效地进行交互。然而，带有CUI的程序会假定用户是人，所以它们与其他程序的交互性并不好。那些设计成能够与其他软件进行交互的软件通常要比设计成与人沟通的软件更为灵活。

你还记得工人用卡车搬家的例子吗？我们曾经说过个头大的家具是没办法彼此拼合在一起的，只有小尺寸的家具才最灵活。同样，程序员们会发现因为其规模，有CUI的程序很难彼此连接在一起。CUI往往会让程序的规模变得庞大。大型程序就如同大件家具一样，并没有很好的便携性。搬运工不会说"能把钢琴递给我"这样的话，同样，程序员在一夜之间也无法将单一复杂的应用程序从一个平台迁移到另外一个平台。

CUI程序无法与其他程序相结合这个缺点迫使人们只能不断增加它们的规模。由于程序员不能依靠与其他程序的系统接口来获得所需要的功能，因此他就必须要为程序自身创建新功能。这种致命的螺旋式开发曲线不断恶化：CUI程序功能越多，规模就越大；规模越大，就越难与其他程序相接；因为它难以与其他程序相连，它本身就必须加入更多的功能。

6.1.5　CUI 没有良好的扩展性

CUI只能在少数情况下工作得不错。通过限制选项数目，就算是没什么经验的用户也能完成复杂的任务。只要不用处理太多的情况，用户通常能够顺利地作出回应。然而，随着提示数量的无限膨胀，人们便很难去回应高达几百个之多的提示信息。

许多Unix系统供应商都会在Unix系统上提供一个很受欢迎的程序adduser（实际上，它是一个shell脚本）。它允许系统管理员通过“用户友好”的CUI来为系统添加新的用户账号。在大多数情况下它的运行情况颇为良好。可是，如果你想一次性添加几千名用户，adduser的问题就会突显出来。

曾经有一所大学决定从另外一种操作系统迁移到Unix。管理员需要将数千名用户账号转移到新的系统。很快系统管理员就意识到，反复运行adduser命令来添加用户一点儿都不省事。他们最终采取了另一个解决方案，将旧系统的用户账号文件移到新的Unix系统。然后再编写一个shell脚本来将该文件转换成类似于Unix口令文件的格式。具有讽刺意味的是，编写shell脚本对文件进行转换比采用adduser命令更节省时间。

6.1.6　最重要的是，CUI 无法利用软件的杠杆效应

由于CUI程序期望着在某些时刻可以与人进行沟通，因此人们便很难将它纳入shell脚本里。要模拟CUI这种与人进行的对话，需要在shell脚本里编写很多代码。在shell脚本中完成这些对话的功能十分繁琐，因此程序员往往会采用小的用户界面程序与用户进行是非选择的交互，其他的则还是需要用户的输入。

由于CUI妨碍了程序与其他程序的互动，因此它们通常只能用来完成自己最初的既定目标，而做不了其他事情。也许你会认为这恰好是“只做好一件事”的例证，可CUI完全不同于那些自身就是一个应用程序的Unix小命令。它不能提供Unix系统具备的命令混搭和即插即用功能。结果就是，就软件的杠杆效应而言，它毫无建树。

没有了软件的杠杆作用，CUI程序（以及它的开发人员）就无法放大其对计算机世界的影响。虽然在CUI程序刚发布的时候，由于其创新性，早期它可能会备受瞩目，不过，随着软件世界中其他人对此程序构思的跟进，它很快就会失去吸引力。

在计算机世界中，软件的进步日新月异，每10年就会有一大批新想法引发行业的重大变革。单一庞大的CUI程序无法适应这个迅速变化的环境。

6.1.7 “CUI 有什么关系？人们都不愿意打字了。”

有些人可能会疑惑为什么我要先讨论这么多关于CUI的问题。毕竟，这些日子以来，不是图形用户界面居多么？我们可以查看核选框列表，点击几下鼠标，为什么还要关心与命令行输入有关的事情？

简短的回答就是，图形用户界面只不过是CUI的可视化形式。而且，以下这些理由可以说明图形用户界面与CUI具有相同的特点。

- 图形用户界面假定用户是人。软件的设计者期望有一位用户出现在计算机前，点击按钮和导航菜单。在某些情况下，设计者还要采取措施来放慢程序运行速度以适应用户的节奏。于是，设计者经常要花费更多精力在制作用户界面上，而不是提供有用的功能。
- 图形用户界面的规模通常都很大，而且难以编写。大型IDE如微软的Visual Basic和Borland的JBuilder 虽然对此问题有所改善，但是，这些IDE本身也有其他问题，我们在别的地方已经探讨过了。
- 图形用户界面往往采取“大即是美”的做法。人们觉得如果5个选项能满足需要，那10个选项的效果会更好。这也是一些办公软件变得规模庞大的原因。当你只想敲几个钉子的时候，完全没必要聘请建筑公司那个配备了钉枪的团队。
- 以图形用户界面为基础的程序很难与其他程序结合在一起。除非一开始这个程序就设计好了要和另外一个程序相接，否则它不会具有良好的接口性。在微软的世界中，OLE和COM控件提供了一些连接功能。对于大多数Unix和Linux命令来说，这种接口完全没有必要，因为它们中的大多数早就已经与其他程序良好地连接在一起了。
- 图形用户界面不具备良好的扩展性。它们有着和CUI一样的问题，如adduser那个例子所示。点击几次鼠标进行操作很容易，可点击鼠标来执行几千次同样的操作则会给用户带来挥之不去的绝望感。这些繁琐枯燥的动作应该由计算机完成，而不是让人来做。

- 图形用户界面无法利用软件的杠杆优势。人们很难通过图形用户界面来执行脚本操作。它往往需要依赖记录程序，靠它们来捕捉鼠标和键盘事件。其实这样做也只是提供了一个不完美的解决方案，因为意外情况经常会发生，用户得根据程序产生的输出结果来作出决定。记录性脚本通常很少有应对这种情况的能力。因此经常需要手动修改脚本，这也就意味着你再一次进入了脚本编写者的世界，而不是图形用户界面的世界。

迄今为止，在这一章中我们讨论了CUI对程序造成的几种障碍。在某些情况下CUI自有它的道理，但这些都是特例而不是常规情况。应用程序如果是小模块的集合，而且模块之间能够彼此沟通合作，那它的售价就可以定得更高。它们是否能够与人类进行良好的交互并不重要。毫无疑问，最终会有一个专门的程序，就像CUI一样来管理这种交互行为。

能够交互的程序实际上就是数据过滤器。通常，每个程序从它的输入流那里收集一些字节，对其应用过滤算法，最后通常会在它的输出系统中产生一些字节。这里说“通常会”，是因为并非所有的程序都会把数据发送到它们的输出流。这要看具体的数据和算法，有的干脆什么也不输出。

事实上，程序过滤数据这个事实很重要。所有计算机和它们的程序都是用来筛选数据的。这就是为什么我们称它们为数据处理器的原因。处理数据就意味着对数据执行过滤动作。

如果程序是过滤器的话，那它就应当表现得像一个过滤器的样子。也就是说，它不应该只是合成数据，而是要选择性地传递那些提交给它的数据。这是Unix哲学下一条准则的本质。

6.2　准则9：让每一个程序都成为过滤器

6.2.1　自有计算技术以来，人们编写的每一个程序都是过滤器

每个程序，不管它简单还是复杂，都以某种形式接收数据作为它的输入，并产生一些数据作为输出。至于程序过滤器如何处理提交给它的数据，则要看程序中包含的算法。

大多数人都认同，文本格式化工具和翻译器这样的软件都可以被视作是过滤

器，但他们无法意识到，那些大家通常不认为是过滤器的程序其实也是过滤器。我们可以拿实时数据收集系统来举例。典型的系统会定期对模数转换器作采样，并从它的输入流中收集数据。然后，它们会筛选数据中合适的部分，并将它作为数据输出流传递给用户界面、其他应用程序或存储介质。

那么图形用户界面也是过滤器么？当然是。通常，图形用户界面会处理按键动作或是点击鼠标这样的“事件”。这些事件形成数据流将作为窗口系统控制下的应用程序的输入。应用程序则作为过滤器来响应这些事件，从而影响到显示屏的变化。

人们会对那些因为硬件错误而出问题的程序有疑虑。假设一个程序在从磁盘读取数据的时候碰到了一个硬件错误。那在这个读取操作中，程序无法获取数据，而只会收到一个错误状态。大多数时候程序都会处理这个错误状态并产生相应的错误信息来提醒用户。换言之，错误状态的输入产生了错误信息的输出。因此，决定产生什么样错误信息的算法便成为错误状态输入的过滤器。

6.2.2 程序不创建数据，只有人类才会创建数据

人们通常认为数据是由应用程序创建的，其实，应用程序根本无法真正产生数据。数据的合成需要创造力，它需要一个原始信息来源，而计算机本身并没有原始信息来源。

当某人在使用文字处理软件的时候，他撰写的文字来自他的大脑。文字处理软件只是一个工具，它采用易于操作的方式来收集和存储想法。它自己无法编写出一本书，就像锤子和一盒钉子也不能盖出房子一样。更智能化的文字处理软件——也就是那些我们称为“所见即所得”（What You See Is What You Get）的类型（或WYSIWYG类型），可能会做一些过滤处理，比如格式化和调整，但它们仍然不是产生最初思想的数据源。

收集我们周围世界信息的实时程序也不创建数据，那些数据早已存在。通过从周边环境提取读数，程序执行了一个筛选的过程。只有那些重要的数据才会通过程序的过滤器，得以保存下来。

世界充满了由人类创造的数据。就算人们从未发明出计算机，这些数据也依然存在。计算机只是使我们能够更有效率地收集和筛选数据。任何由此产生的“新”数据其实并不新。计算机只是给了我们一个机会，以不同的方式来管理这些数据。如果这个世界上到处都是计算机而没有人的话，就不会有任何数据。

6.2.3　计算机将数据从一种形式转换成另一种

换一个角度看，对应用程序有用的数据必须以一种便于操作的格式保存下来。例如，对任何一个软件而言，人们敲击键盘的动作其实没什么价值。然而，一旦这些动作被转换为代表着一系列二进制数据的电子脉冲，它们便在机器内部得到了新生。然后，软件可以将这些数据转换成各种形式以满足各种各样的需求。

还有一个有趣的值得注意的事情就是，音乐也是一种数据。它以这样或那样的形式存在了好多个世纪。几千年来，人们一直在用弦乐器演奏。钢琴就是一款弦打击乐器，其历史长达好几个世纪。钢琴家们用很多卷打孔纸来“记住”歌曲的乐谱。近年来，键盘合成器成为了采集音乐“数据“的主要手段，音乐家们移动手腕与手指所产生的音符都被其捕捉。。一旦数据被捕获之后，它就被转换为音乐产业中的标准电子音乐格式——MIDI文件。有了MIDI文件，人们就可以通过无数方式对数据进行过滤，它不仅能够转成钢琴的音调，还可以产生其他任何超出自然声学乐器功能的声音。在许多方面，音乐世界中MIDI文件的使用仍然处于起步阶段。未来MIDI的应用将会带给我们一个神奇的音乐之旅，延伸人类对声音的感知度。这些现象之所以会发生，是因为如今的程序可以采用从前不可能的方式来过滤音乐数据。我们最为肯定的就是，计算机将会修改并完善MIDI数据，以产生创造性的全新声音。

6.3　Linux环境：将程序用作过滤器

你可能想知道将程序作为过滤器使用意味着什么。Linux程序员遵循着一些不成文规则，从而简化过滤器软件的设计过程。为了澄清这些规则，我在这里已经涵盖了若干条准则。但是，在讨论这些之前，我们很有必要先来简短地说明一下Linux上的stdio（标准输入输出）概念。

Linux系统调用某个程序时，通常会有三个标准I/O通道对其开放，也就是所谓的stdin（标准输入）、stdout（标准输出）和stderr（标准错误），这就是stdio得名的原因。至于连接在这些I/O通道另一端的事物会是什么，则取决于程序是如何调用的。默认情况下，stdin收集用户的输入，而stdout则将输出结果发送到用户的显示屏幕。任何发送到stderr的输出也会同时在显示屏幕上出现，但这些数据通常被人们认为是带外（out of band）信息或错误信息。

Linux的stdio有一个有趣的特点，连接到I/O通道的设备并没有硬性地与程序相连。在调用程序的时候，用户可以指定输入数据来自何方，或是该把输出数据发送到哪里，不一定非要发送到用户终端。举个例子，stdin可能来自一个文件，或是通过Linux管道相连的其他程序，甚至连接世界另一端的卫星式系统。同样，如果用户想把stdout的输出结果保存在一个文件中，备以后查阅，他便可以配置shell的环境变量来指明将输出存储在哪里。因此，这种做法对数据的源和目的地来说，都提供了巨大的灵活性。

Linux程序员如何处理stdio会对程序作为一个过滤器的运作能力产生重大影响。如果程序编写正确，所有stdio的适应性都可以为我们所用。否则，用户很可能被锁定在图形用户界面里。那么，关键就在于正确编写软件以采用stdio的特性。这里面有以下三个重要的原则。

(1) 使用stdin作为数据输入

那些从stdin获得输入数据的程序，总是假定它们的数据会来自任何地方。事实上，这是对的。通过避免硬性指定输入通道，在程序调用的时候，用户就可以轻而易举地指定输入来自何方。输入数据的来源可以是键盘、文件、其他Linux程序，甚至于图形用户界面。

(2) 使用stdout作为数据输出

正如同stdin允许程序接受来自任何地方的输入数据那样，stdout也允许程序将输出结果发送到任意地方。在这里“任意地方”可能是指用户的屏幕、文件、打印机，甚至是数字语音合成器。选择权在于用户，程序运行时可以将数据输出到任何适当的地方。

(3) 使用stderr处理带外信息

错误消息和其他的警告应该通过stderr发送给用户。它们不应该出现在stdout的数据流中。其中一个原因是，用户可以选择使用单独的文件来捕获错误信息或者立即在终端上查看。将错误信息发送到与stdout相同的I/O通道会导致后续流程的混乱。请记住，这是因为在Linux上我们很少单独使用某个命令。

大家要注意这个做法不同于其他操作系统。大多数运行在其他系统上的应用程序倾向于硬性实现一切。这些程序的设计者认为总是有用户坐在键盘前。程序可能

会询问用户是否想将输出结果发送到一个文件，但很少会提供这种我前面提到的做法，除非程序员有意识地作出努力将此功能包含在应用程序中。

硬性实现I/O接口暗示着你非常了解程序的所有可能用户。这纯粹就是狂妄自大的表现。在前面的章节中我们强调过，每个人都有自己的学习曲线。没有人能够预测他的软件最终会被如何使用。你能做到的最佳效果就是让程序的用户界面尽可能灵活，能够处理今天存在的许多不确定情况。明天的事情就让未来的程序自己去搞定吧。

在软件工程领域里工作多年之后，我曾经担任过一个电话支持中心的职位，在那里我需要解答客户关于软件的问题。在许多情况下就是我自己开发的软件。这份工作经历给我的启发最大，因为我直接与那些在日常工作中使用软件的客户进行交流。我认真听取了几百名客户讲述的关于他们如何使用我的软件的故事，那真是令人难以置信。我最常见的反应就是："我的软件不是用来干这个的!"

最近一些网络非法入侵现象已经很清楚地表明了一个令人痛苦的事实，恶意用户经常会超出软件原本设计意图去做各种令人意想不到的尝试。这些非法入侵通常会导致缓存溢出和其他的编程错误，让系统变得脆弱无比。虽然程序员的意图也许是程序该以某些方式来运作，但系统黑客却会找出一些稀奇古怪的做法，而且往往采用不怎么光彩的方式来使用它们，从而获得未经允许的访问权利。

你永远猜不到人们将如何使用你的软件。永远不要自以为是地认为他们会完全遵从你的设计意图。你可能认为自己正在编写一个简单的排序程序、文字处理软件或是文件压缩程序。但很快你就会发现有人却在用排序程序将ASCII码转换成EBCDIC码，而字处理器却已经变成了公共访问的公告板，文件压缩程序正在用来把本地有线电视系统播放的电影《飘》数字化以供下载。

如果你时刻谨记所有的程序都是过滤器，那在开发过程中就更容易避开图形用户界面。当你假设可能是另外一个程序而不是由人来接收程序的数据流时，就能消除我们固有的偏见，即试图让应用程序具有用户友好性。你不会再局限于思考菜单的选择项，而是开始研究数据最终可能会到达的地方。尽量不要把注意力集中在程序可以做什么上面。相反，你应该考虑一下程序的发展方向。然后，你就可以纵览大局，发现这个程序只是其中的一个小部分。

当软件设计师将程序视为过滤器的时候，就可以把应用程序分解成更小的程

序，每个程序只执行应用程序的一个功能。这些小程序不仅互相沟通良好，而且没有那些花里胡哨的号称能使软件用户界面“绝对安全”的功能。从某种意义上来说，它们的“用户”其实就是其他程序。这些程序最终能产生更高性能。未来，在更快架构推出的时候，这些小程序不会受到人类生理机能的限制。而且，正如我们以前说过的，程序从来不会抱怨，完全没有懈怠情绪，也不需要请病假。

现实生活中，如果程序员也能不抱怨、不懈怠或是不请病假的话，那会多美好！

第7章 更多Unix哲学：十条小准则

至此，我们已经探讨了那些构成Unix哲学核心的准则，它们是Unix世界的坚实基石。任何对这些准则怀有强烈异议的人都无法理直气壮地宣称自己是“Unix人”。即使他们觉得自己就是，也会招来Unix社区（Linux社区也包含其中）的普遍质疑，被认为缺乏对Unix及其理念的真正信仰。

在经受过Unix“宗教教条”式的训导之后，我们准备着手阐述其中的一些理论教义。Unix和Linux开发人员不遗余力地维护着我们所讨论过的准则的完整性。而另一方面，人们对这些准则也许还是会有一些认同感。虽然不是每一位“Unix人”都完全同意本章谈到的观点，但就整体而言，Unix社区（如今的Linux社区）普遍遵守着这些原则。

你会发现本章谈及的某些重点事项更侧重于我们应该怎么做，而不是为什么要那样做。我会尽量做一些解释，但你要知道有些事情其实本就没道理可言，只是一贯的传统做法而已。就像宗教一样，Unix也有它自己的传统，而Linux则以一种“新瓶装旧酒”的方式表现出同样的特征。

很明显，我们并没有过多地谈及开源，这可是大多数Linux“意外革命者”奉为圭臬的东西。因为开源精神在Linux文化中根深蒂固，早已成为如Richard M. Stallman和其他先驱者的战斗口号，所以，只将它作为一个章节中的小插曲来讨论是不够的，稍后我们会对开源做更深入的探究。

其他操作系统的支持者也认同其中一些Unix哲学的小准则，这并不奇怪。好的想法往往能传播到整个计算机世界。其他系统的软件开发人员会发现，某些Unix概念表面上看起来并不适用，但实际上却很有价值。于是，他们会把这些理念融入

到自己的设计中，甚至有的时候这些系统和应用程序都会有带有Unix的影子。

7.1 允许用户定制环境

多年以前，我曾为X Window System编写过一个叫uwm的窗口管理器，也是一款用户界面。它给用户提供了很多在今天的窗口系统中被人们视为理所当然的功能：移动窗口、调整大小、改变堆叠顺序等。这个程序得到了广泛好评，后来成为了“第10版X Window System的标准窗口管理器”。它提出的概念在当时鲜为人知，可今天却成为窗口管理器的概念鼻祖，被广泛应用于X Window System之上。现今的窗口管理器大量借鉴了uwm及其衍生产品中原始理念。

Uwm赖以成功的原因之一要归结于Bob Scheifler，也就是他曾在麻省理工学院举行的X Window System早期设计会议上发出的一句感慨。Bob是X Window System整体设计的一个杰出贡献者，当时他正在审阅我为“Unix窗口管理器”撰写的几页设计规范，突然他脱口而出：“假如我不想用鼠标左键来做那个呢！”他接着建议，也许用户可能会喜欢自己定制每个鼠标按钮的初始功能。

如果我是一个漫画人物，你就会看到当时我脑子里的那个灯泡一下子亮了起来。

uwm的开发沿袭了这个思路，并在定制用户界面领域开辟了一片新天地。X Window System本来就可以让用户选择自己的窗口管理器。而uwm程序又在用户定制上更进一步，允许用户选择窗口管理器的外观、感觉和行为，甚至鼠标的移动、按钮的点击、颜色、字体、菜单选项都可以由用户自行决定。这个概念是如此强大，以至于X Window System的开发人员后来还设计了一款特别的资源管理器，它几乎能让用户控制屏幕里每一个元素。这种前所未有的灵活性到今天仍未逢敌手，即便微软和苹果的桌面环境也无法企及。其他系统只将如此灵活定制的权力赋予了开发人员，而X Window System却将其提供给了普通用户。

早些时候我们曾经说过，人们在某些事情上投入越多，就越希望获得更多回报。在观察人们如何使用uwm的时候我发现，如果有机会去调整使用环境，那人们会很乐意那么做。内置的灵活性使得用户可以更加投入地学习如何使用某个应用程序，从而获得最大收益。随着人们在量身定做的环境中变得如鱼得水的同时，他们就会愈发难以适应那种没有定制功能的环境。

今天，大部分Linux环境都遵循着这个基本理念。在刚开始使用Linux的时候，人们普遍会觉得这真是个烦人的系统，因为它太灵活了。过多选择反而让大家无所适从。从一开始Linux就提供了各种各样的选择：众多发行版、多种窗口管理器、多个桌面、多款文件系统等等。你也用不着只盯着一家厂商购买Linux操作系统。事实上，你甚至都不需要浪费钱。任何一家供应商网站都提供免费下载的Linux系统。

最终，用户还是找到了适合自己的Linux操作方式。他们选好了发行版，并用心学习该如何最大限用地利用系统中的诸多功能。一旦他们倾注在这个系统上的时间和精力达到一定程度，切回到其他操作系统就变得困难无比。他们对Linux的热情之大以至于他们宁愿改变自己去适应原本不喜欢的事物，而不是完全弃用之。

有些人对Linux颇有微词，抱怨它迫使用户先要投入大量时间和精力先学习，然后才能真正有效地利用好Linux。他们认为，使用Linux很容易“搬起石头砸自己的脚”。这也许没错。但是Linux的倡导者Jon“maddog”Hall曾下过论断，搬起石头砸自己的脚总比裹足不前要好。

7.2　尽量使操作系统内核小而轻量化

这是Unix纯粹主义者的一个热门话题，多年以来也成为大家辩论的主题。Unix内核包括一些例程，它们基本上只管理内存子系统以及与外围设备相连的接口，不做其他事情。任何时候只要人们想得到更高效的应用性能，他们第一个建议就是将应用程序放在内核中运行。这样可以减少程序运行时在存储空间上下文切换的次数，付出的代价则是内核规模变大，并且不易兼容其他Unix内核。

在X Window System的早期开发阶段，人们产生过强烈分歧，讨论将X服务器部分例程嵌入到Unix内核中是否能产生更高性能。（X服务器是X Window System的一部分，用来从鼠标和键盘事件中捕捉用户输入，并在屏幕上呈现对应的图形对象。）X服务器运行在用户空间（即内核之外的存储空间），这使得它相对更具有可移植性。

“将应用放在内核”阵营提倡，要充分利用这种减少内核和用户空间上下文切换次数的优势来提高性能。他们的理由是，内核规模的迅速增长不会有什么影响，因为相比早期的系统，现代Unix拥有更多可用内存。因此，还有充足空间去满足应用程序运行的需要。

另一方面，“将一切放在用户空间”阵营却认为那样将使X服务器变得不可移

植。而且，修改X服务器的程序员不单要成为合格的图形软件开发人员，还需要变成Unix内核高手。可是，成为Unix内核高手的代价不菲，开发人员必定得放弃自己在X服务器图形领域中的一些兴趣，X服务器的开发工作便会受到影响。

那么，两大阵营最终是如何解决这个分歧的呢？系统自己作出了选择。在貌似成功地将X服务器移入内核之后，测试人员发现X服务器的bug不仅会使X Window System崩溃，甚至还波及整个操作系统。操作系统崩溃比X服务器崩溃还要严重，因此今天大多数X服务器仍然驻留在用户空间里。“将一切放在用户空间”阵营得了一分。

只要能抵制住将一切放在内核空间的诱惑，保持内核小而轻量化就会容易得多。在启动任务时，通过降低数据复制或修改的次数，小巧轻便的内核还是能够加快任务在用户空间的激活速度。这最终使Linux能够更为简单有效地将那些“只做好一件事”的小程序集合在一起。将单一功能的小程序快速激活对Unix整体性能起到了至关重要的作用。

7.3 使用小写字母并尽量简短

对Unix系统而言，人们首先注意到的一个特点就是它采用的几乎都是小写。Unix用户不需要使用大写锁定键来进行操作，所有输入采用小写字母即可。

这么做有两个原因。首先，小写字母看起来更轻松。如果一个人必须长时间与文本打交道，那就能明显地感觉到小写文本看起来比大写的舒服很多。第二点也是更重要的一点就是，小写字母有向上伸和往下伸两种形状，也就是从字母主体延伸出来的向上或向下的细线，从字母“t”、“g”、“p”就可见一斑。在你阅读的时候，它们能够给眼睛传递一些智能化提示，让阅读变得更为轻松。

Unix是区分大小写的。例如，“MYFILE.TXT”和“MyFile.txt”并不是同一个文件。Unix常用命令和文件名采用的都是小写。大写通常用来吸引人们的注意。例如，在目录中将文件命名为“README”就是一种视觉提示，让用户在进行别的操作之前先来阅读此文件内容。此外，`ls`命令在列出目录文件名的时候，通常都是按字母顺序排列的，这样大写的文件名就排在文件列表的前面。人们就会更多地注意到这个文件。

对于长久以来已适应了其他不区分大小写的操作系统的人们来说，在初次接

触Unix的时候往往会无所适从，可最终他们还是能够适应，许多人甚至喜欢上这个特性。

Unix的另一个怪癖就是它的文件名通常都很短。常用命令的长度一般不超过两或三个。Unix奉行简约至上，你会发现诸如ls、mv、cp等这些并不直观的命令。拥有多个长单词的命令往往会使用首字母缩写的形式。例如，“parabolic anaerobic statistical table analyzer”（抛物厌氧统计表分析器）会简略缩写成“pasta”。

使用简短名称的传统由来已久。最早，Unix是在使用电传打字机（teletype）的系统上开发的，并没有CRT终端。在使用电传打字机输入信息的时候，每当人们按下一个键都会伴随有哒哒的提示音，最快的打字员每分钟约能输入15~20个单词。因此，采用简短名称来描述事情便成为流行做法。事实上在过去，大多数不会打字的计算机程序员拿着计算机根本就做不了什么事情（好吧，也许还是能做一点点）。

为什么Linux用户仍然坚持着这个传统呢？毫无疑问，今天的PC键盘让人们大大提高了输入速度，所以简洁性就不再是必要条件。但这么做的原因是，采用较短的名称可以在一个命令行中包含更多命令。你应该还记得Linux的shell有管道机制，它可以把一个命令的输出作为另一个命令的输入。

这个强大的功能在Linux用户中非常盛行。他们经常将很多命令串在同一个命令行里。如果名称过长，他们使用的窗口就会显示不下这行命令。解决方案就是尽量采用简短的命令名，因此当你在屏幕上打开几个相邻窗口的时候，较小的窗口就可以包括更多的命令。

当然，你也许会想：“有这个必要么？”为什么不直接点击鼠标来执行呢？如果你正在反复使用某个命令，这没问题。但要记住，Linux和Unix之所以这么强大是因为它们可以将多个命令动态结合起来去构建新的应用。而今天流行的桌面环境却做不到这一点。WYSIWYG（what you see is what you get，所见即所得）这个词其实也意味着WYSIATI（what you see is all there is，所见即一切）。点击鼠标能完成的工作毕竟很有限，假如人们非要尝试着去提供一个图形化shell，那充其量也不过就是一款繁琐僵化的用户界面。就算使用“快捷方式”也只会缩短运行单个命令所花费的时间。[①]

① Linux上的各种桌面环境都可以让你采用快捷方式来执行多个命令。但是，如果没有这种动态结合多个命令的能力，它也不过是僵化的技能。未来它是否会继续保持这种僵化模式，还有待观察。谁也不知道在创意非凡的Linux人的头脑中，潜藏着什么奇思妙想。

7.4 保护树木

之所以会出现Unix大师这个词大概是因为Unix就是这样一个非比寻常的操作系统。人们对那些Unix专家有着一种又敬畏又怀疑的矛盾情结：敬畏是因为他们掌握着该奇秒计算机环境的奥秘，而怀疑则是因为你不知道这些人脑袋是否有点儿坏掉，在一个繁琐操作系统上才会投入这么多宝贵时间。

我遇见的第一个Unix大师是我的老板，那还是职业生涯早期在新罕布什尔州南部某小型高科技公司的时候。除了防止我们在上班时间偷玩“盗贼”（一种流行的地牢游戏）外，他也曾肩负着公司Unix系统日常管理责任。我们的Unix系统运行在DEC的PDP-11/70机器上，在他到来之前，这台PDP-11/70运行的是另外一款操作系统。

那些日子里，大部分编程活儿都是采用汇编语言完成的。我们部门的工程师需要花费大量时间编写和测试代码，并将它们交叉编译以运行在不同体系结构的目标机上。打印简单的“位运算”程序都需要耗费很多张纸。为了调试程序，程序员往往会生成一份汇编列表并将其发送给运行最快的行式打印机，有时候打印出来的文件厚达6英寸。程序员越资深，他的文件就越厚。如果有人想受到部门全体成员的尊重与爱戴，他只需要拿出一堆厚得无人可及的文件即可。这么厚的文件肯定意味着工作的卖力程度，在绩效考核时他便能得到更丰厚的薪酬回报。管理层对这个神话深信不疑，我们这些工程师也知道该如何玩转这套“潜规则“以获取相应回报。

一天，我正抱着一大堆厚达5英寸的编程“副产品” 走在大厅里，我的老板，这位Unix大师拦住了我，问我这么厚一叠纸都是用来干什么的。“这是我的程序，”我回答道。几乎要凑到他鼻子下的文件仿佛在说：“是的，我在努力工作。先生!”

他做了个鬼脸：“看你毁了多少树木。来一下我的办公室吧。”

他径直走向他的机器终端，给我上了一堂关于Unix的课，这堂课令我永生难忘。

他阐述的重点是：一旦把数据打印在纸上，你就基本上失去了进一步操纵它的能力。纸上的数据无法进行排序、移动、筛选、转换、修改，或是其他任何可以在计算机上轻而易举地完成的事项。你无法敲几个按键就对它的内容进行检索。你也无法将它加密以保护敏感信息。

你还记得早些时候我们曾经提到过的一个概念吗？不动的数据成为了“死”数据。打印在纸上的数据也是如此。移动纸本文件根本不可能像移动存储在计算机上

的电子数据那么快。因此，相比电脑里保存的数据，纸上的数据永远是“过时”的。你只需问问那些出版过流行百科全书的出版商就会了解。在人们可以从互联网上获得更多及时信息的今天，他们正在为该如何兜售那些外观可爱、采用皮革装订的精装百科书而愁肠百结呢。

随着传真机日益普及，你可能会观察到通过电话线来传送文件也变得轻而易举。但问题是，传送完毕后的纸本信息命运究竟如何？纸质媒介将它困住，并大大地限制了它的功效。

请谨慎使用纸张。它就像是为数据开具的一份死亡证明书。

7.5　沉默是金

当然，我们并不是说Unix程序员应该避免使用多媒体。我们谈论的是那些所谓的用户友好程序，它们往往喜欢夸大事实或是对用户过于热情，好像他们是该软件的密友。太多程序员认为应用程序该采用一种“话痨式“交谈风格，这会对用户大有帮助。而Unix却在这个方面显得异常沉默寡言，因为它提供的“只是事实”，仅此而已，不多也不少。

在没有收到任何输入数据或是没数据可输出时，许多Unix命令会保持沉默。对Unix新手来说，这么个状况可有点令人忐忑不安。例如，在常见的非Unix系统上，当在一个不包含任何文件的目录下输入以下命令时，请注意系统是如何通知用户没有发现任何文件的：

```
$DIR
DIRECTORY: NO FILES FOUND
$
```

比较一下Unix的ls命令对上述情况作出的回应。当它在一个目录中找不到任何文件的时候，它只是返回命令提示符：

```
sh> ls
sh>
```

许多用惯了非Unix系统的人经常批评Unix没能通知用户目录是空的。另一方面，Unix的拥趸却认为ls命令拒绝打印任何信息实际上就等于告诉了用户，目录中没有发现任何文件。文件名的缺失本身就能证明该目录为空。就好像我们通常会说

“房间是黑的”，而不是“房间里面没有任何光线”。这其中存在着只是非常微妙的差异，然而，这个微妙差异却很重要。

那么，在没有数据的时候，命令仍安静运行到底有什么好处呢？一方面，屏幕上只会包含有效数据，而不是充斥着杂乱的注释，它们几乎传递不了什么有用的信息。如果没有连篇累牍的废话，我们就能轻而易举地找到不太起眼的真实数据。

虽然这种做法是为了保持简洁性，但其中还有一个更具技术性的原因。正如我们前面讨论过的，大多数Unix命令经常被当作过滤器配合Unix的管道机制一起使用。例如：

```
ls -l | awk '{print $4}' | sort
```

-l参数会使ls命令产生一份更长更详细的文件清单。管道符号“|”将ls命令的输出传递给awk命令作为输入。而'{print $4}'这一部分则让awk命令只打印ls每一行文本输出结果的第4个字段，并丢弃其余部分。系统将此字段传递给sort命令，它会对输出结果按字母顺序进行排序。

通常情况下，当该目录包含多个文件时，一切显示都很正常。但当该目录为空时会出现什么情况呢？由于ls命令没有输出，管道便断裂了，awk的进一步加工和排序动作不会发生。但如果ls输出一条像“DIRECTORY：NO FILES FOUND”（目录中没有发现任何文件）的信息并传入进管道，这会导致sort命令的输出结果中出现一个奇怪字符“FOUND”，这是因为“FOUND”的的确确是输出结果的第4个字段。也许ls不明确发出警告提示用户这是一个空目录的行为算不得“用户友好”，但它设计主旨是为了既告诉用户该目录为空，同时也能让ls命令用得上管道机制。

一位Unix系统管理员曾经指出，系统日志才是他真正的救星。而且他还提到，如果没有这些额外信息，他便无法解决碰到的一些配置问题。在需要额外输出信息用于调试的情况下，你只需要将错误信息重定向到日志文件供日后审阅即可。

在Unix环境中，重要的是言简意赅地传递你要表达的意图，仅此而已。

7.6 并行思考

计算机世界里有个流传已久的老笑话：如果一个女人要花9个月时间生下一个婴儿，这是否意味着可以让9个女人在一个月内生出一个婴儿？笑话中蕴含的道理

很明显，自然天性导致某些任务必须串行执行。任何试图让其并行运行的举动无法加快它的进程。

除掉生育之外，当今世界上还是有许多流程是可以同时进行的。建筑施工队、电视制作团队和专业篮球队就是团队实体必须并行操作的例证。为了完成目标，他们会同时执行一系列操作过程，每一个成员都需要一部分任务。在执行完自己的任务后他们会碰个头，以确保各自进展能够满足既定目标，就像计算机应用程序使用信号量（semaphore）和其他进程间通信机制一样，目的都是让自己的状态保持同步。

就Unix来说，并行思考通常意味着你应该尽可能多地利用CPU的运算性能。今天，大多数CPU表现远远超出了其他大容量存储硬件，比如硬盘、软盘，甚至是内存芯片。为了充分利用系统资源，你必须让处理器保持忙碌的工作状态，这样它就不会闲置等待其他外设。

Unix的做法是同时运行多个进程，每个进程完成一部分任务。那样的话，就算有进程被外围设备阻塞，还是会有一些进程仍然在运行，从而大大提升效率。因此，如果以总体完成的工作量作为评判标准，在相同硬件平台上Unix系统的表现往往要优于其他操作系统。

并行机制对用户如何看待你的应用程序也产生了巨大影响。Unix用户一直很喜欢阅读来自世界各地的文章集合，即网络新闻（net news），它通过互联网和Unix之间点对点的复制程序uucp发送。这些文章涉及的主题范围非常广泛，每天都会出现成千上万的新文章。网络新闻和其他类似的论坛就好像人们开电子讨论会议的场所。

网络新闻的一个问题是所谓的“新闻组”往往占据着系统上的多个文件目录，并包含成千上万文件。打开如此庞大的目录来获取文件和它们的标题列表往往缓慢且繁琐。过去的软件通常需要好几分钟才能打开一个大型新闻目录。等待的过程让用户颇为沮丧。人们很难从一个新闻组迅速地跳转到另外一个新闻组。

于是，Larry Wall编写了一个叫rn的程序。它像其他程序一样读取新闻组目录的内容，不过它有一个特别之处：在用户阅读新闻的时候，它会在后台获取目录的文件列表，而且会在用户毫不知情的状况下悄无声息地执行着这个操作。区别就在于，当用户决定要切换到一个不同的新闻组时，rn命令其实早已预读完了此新闻组的内容，内容可以立刻显示。这就像提前预定比萨一样，当你到达目的地时，比萨

也早就送到了。

对rn的用户界面来说，并行具有明显优势，它同样也能在其他类型的软件中发挥巨大影响。例如，Unix允许用户在命令后面添加一个“&”字符，从而让该命令在后台运行。如此调用的命令启动了一个与命令解释器或shell并行的进程。它的效果就相当于同时执行多个任务。通过保持CPU的繁忙运作，而不是让它闲置在那里等待外围设备完成I/O请求，可以大大提高机器的使用效率。

关于并行思维，我们最后要牢记的一点就是：不管机器速度快慢，你都可以将多个机器串联在一起，从而得到一个运行速度更快的集群。由于CPU芯片的价格还在持续下跌，未来，在更为理想的状态下，系统中会有数百、数千，甚至数百万个CPU处理器。将单一功能的小程序逐一分配给各个处理器，这样我们就有可能完成系统当前无法胜任的任务。Unix早已确定了自己在集群领域的领先地位，Linux也有望主宰这个领域。

7.7 各部分之和大于整体

古老的羊角锤是木匠业的惯用工具，钉子出现有多久，它的历史就有多久。羊角锤可以用来敲钉子和拔钉子，这个单柄木头工具结合了两种实用功能。此外，虽然有些锤子改善了手柄或是转而采用钢爪，但它们的基本思想仍然不变。它经受住了时间考验。

或者，真是如此吗？

今天，最专业的木匠仍在使用羊角锤，不过它还是受到了新技术的威胁。羊角锤已不再是木匠们的最佳利器，它只是一个方便的替补工具。比如，射钉枪敲钉子的能力就已远超它。射钉枪能够快速完成工作，准确率也更高。有了射钉枪，工人建造房屋所用的时间比过去要少得多。但是，这些工具前爪端的样子并没有什么变化。看来对通用起钉工具而言，高科技也改变不了它的基本概念。

有些人的软件就像这个锤子。它提供多种功能，能够紧密结合在一起轻松完成任务。问题是，很多集成性应用是大规模单一化的解决方案。当然，它们能够完成工作，但它们会让用户甚至系统本身都不知所措。这个结合了太多功能的系统，让人难以理解，更不用说让普通用户轻松使用了。

Unix的做法是将小型组件集合在一起，构建集成性的应用程序。这样，就可以只加载和使用所需功能。它同时还保留了灵活性，可以对应用程序中的部分功能作选择性修改，而不需要去替换整个应用程序。你可以参考一下图7-1。

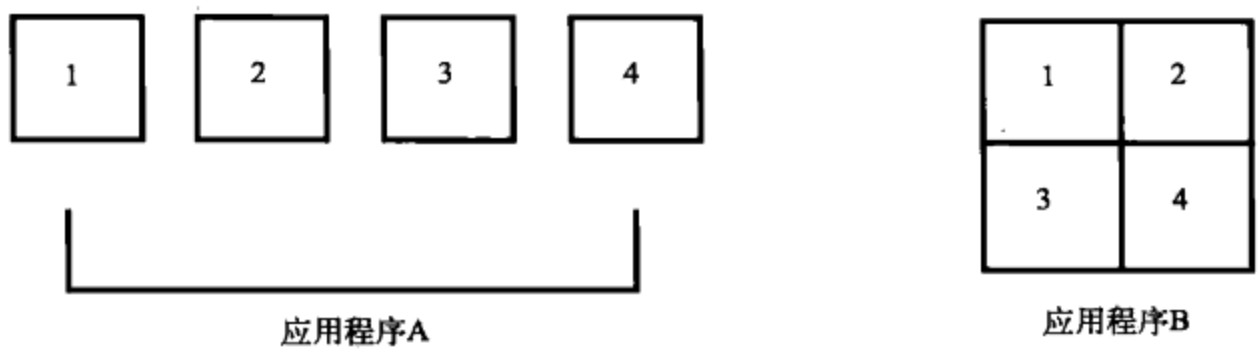


图7-1　对比应用程序A与应用程序B

应用程序A和应用程序B都包含相同的功能。就这个层面而言，它们是等价的。当4个功能都需要的时候，它们有着相同的工作效果。然而，应用程序B是一个大型的单一程序，而应用程序A却是一个小程序的集合，其中的每一个小程序分别提供了应用程序B中包含的部分功能。

在需要使用所有应用功能时，这两个应用程序几乎没什么差别。不过，假设你想要一个只包括功能1和2的新应用程序。对于应用程序B而言，你不得不负担B所有模块的资源开销，哪怕只需要利用其中的一半功能。而另一方面，应用程序A就只需要将功能1和功能2捆绑在一起，然后方便快捷地构建一个新应用程序。

再假设你想要一个新应用程序，其中包含功能2、3以及额外的功能5。应用程序A又可以轻而易举地完成这项任务：你只需要创建一个功能5的新模块，然后抛弃功能1和功能4。修改应用程序B就复杂多了，软件开发人员需要对它做个大手术。这本来应该不是问题，但这种单一程序往往难以修改。程序员在构建大规模、综合性应用程序的时候，经常会为了获得更好性能而牺牲扩展性。取舍的后果通常会使得代码繁琐复杂，不管谁来维护，我敢肯定这个人都会头痛不已的。

我们还要注意，应用程序B不允许在各个功能之间使用管道，而应用程序A却可能在很大程度上依赖着管道。这样也就不需要因为单个功能的改变而去重新链接应用程序。这也加快了开发和测试进度，因为维护应用程序A的程序员可以只处理那些易于管理的更小组件。

应用程序A还有一个不易察觉的优势，它更利于多个开发人员的并行工作。当

程序员知道他们是在处理那些必须互相沟通的独立程序时，他们就会重点处理好模块间的接口。这有助于消除杂乱冗余的代码。

你可能想知道用户该如何与应用程序A中的各个组件进行交互。其中一个解决方案是，将独立的用户接口模块纳入到应用程序A中。由于应用程序A的模块已设计成可替换的形式，这样才有可能为不同类型用户定制不同用户接口模块。 X Window System就是一个实际例子。它提供了多种选择，用户可以随心所欲地挑选自己喜欢的用户界面。

7.8 寻求90%的解决方案

当像联邦快递（Federal Express）公司这样的隔夜快递服务首次出现在美国的时候，人们对它极高的工作效率啧啧称奇。想象一下，不用登上一个月或一星期，一夜之间他们就能像快马一样将你的包裹送达全美各个角落。对许多人来说，这就是梦想成真。这种快递服务彻底改变了当代商业模式。

联邦快递和其他隔夜快递服务公司的兴起引发了一个严肃的问题：为什么美国邮政服务公司（U.S. Postal Service）无法提供这项服务？我们都知道，今天美国邮政服务公司能够并已经在提供隔夜快递服务。尽管如此，许多人仍然坚信商业公司的包裹递送服务表现更佳。为什么呢？这是因为那些商业化服务公司只提供了90%的解决方案。

完成90%的事情总比做到100%来得容易。就拿联邦快递来说，它没有在内陆穷乡僻壤地区提供与大城市区域类似的业务。在全美主要城市的每一个街角，联邦快递的包裹运输车几乎随处可见，但让你在蒙大拿州山区的偏远小镇找找看呢？对于联邦快递而言，偏远地区的服务可以说是无利可图，所以它不在这些地区开展业务。但作为美国的政府机构，美国邮政服务公司却必须为所有美国公民提供同等服务。因此，你会发现几乎每一个美国小镇的街头都有邮政信箱。

在遵从政府法令的前提下，美国邮政服务公司必须提供100%的解决方案。而与此同时，联邦快递作为一个独立经营的载体，却可以专注于高利润且容易做到的90%优化解决方案。结果呢？联邦快递能够以较低价格为企业和消费者提供隔天送达（有时甚至在当天）服务。可美国邮政服务公司却要满足所有人的需求，才受限于这些低效率的运营条件而表现得差强人意，可以说它只是勉强完成了它份内工作。

90%解决方案的精髓在于，我们需要故意忽略那些代价昂贵、费时费力或难以执行的项目。如果有机会将那最棘手的10%舍弃，你肯定可以轻而易举地解决世界上的大多数问题。

敏捷Linux和Unix软件开发人员在设计某个应用程序的时候，我们可以这么说，他们会力求提供一种“低投入，高回报”的解决方案。这便意味着他们会毫不留情地砍掉那些很少有人使用或是实现起来非常费劲的产品需求，而且他们的态度往往很坚决：“如果真的有人需要这些功能，就让那些人自己动手吧”。

诚然，有时候90%的解决方案是万万行不通的，比如心脏移植手术。不过这些情况在计算机世界里相当罕见。请记住，大多数软件只是多方妥协后的成果，从来没有完成的时候，它们只是会被发布出来。如果按照严格定义，软件是完不成的，没有人能开发出一个提供100%解决方案的软件。假如我们已经认识到完成90%的解决方案算是一个合理标准，那就编写一个能够吸引大多数用户的应用程序就是小菜一碟。

这就是Unix赖以成功的部分原因。它并不奢求为所有人提供“一揽子”解决方案，而只是满足大多数人的需求。剩下那些人，完全可以让他们去编写自己的操作系统。

7.9　更坏就是更好

在军队中呆过的人都知道有3种处理事情的方法：正确的方法、错误的方法和军事的方法。

正确的方法就是那种所有人一致公认的好办法。它的做法方方面面都完美无缺，没有任何争议。它是恰如其分、量身定做的方案。

错误的方法则是正确的方法的逆向操作。它错得非常彻底，在任何人的眼中它都是大错特错。所有人都一目了然。

迄今为止，3个方法中最神秘且最有趣的就是这个军事的方法。正确的方法与错误的方法是互逆关系。而军事的方法却如同裹了一团迷雾，它既不黑也不白。看起来它应该是一种失败的做事方式，可是出乎意料，本该惨遭失败的事情却会在不知不觉中获得空前的成功。

Unix的做法有点儿类似军事的方法。如果你都听从纯粹主义者的意见，那Unix早在20年前就可能消亡了。尽管饱受批评者的非议，它还在蓬勃发展，而且一天天地壮大。

在Unix的做法中有个根深蒂固的矛盾概念，即“更坏就是更好”。许多人声称Unix不如这些或那些系统，因为它的用户界面实在有点儿糟糕，Unix简单得不像是个正儿八经的操作系统。Linux也曾因为它缺乏适当的图形用户界面而一度饱受恶评，而且一度还难以安装，并缺乏一款好用的办公应用软件。

然而，如果Unix真的在很多方面表现得比大多数系统差，那么它就愈发证明了“更差的”事物生命力反而比那些“正确”或“错误”的事物高。在一个新技术层出不穷，现有技术转瞬过时的世界中，Linux和Unix均表现非凡，能坚持到底。

计算机世界中存在一种主流思想学说，认为良好的都设计应该具备4个特点：简洁性、正确性、一致性和完整性。在能被观察到的层面上，设计应该是简单、正确的（没有bug），它的主体思想应当贯穿始终并具备完整性，也就是能够涵盖人们预期到的所有合理情况。

大多数Unix程序员都会同意这样的观点，应用程序和系统就是应该具备简洁性、正确性、一致性和完整性。关键是他们该如何决定这些特性的优先级。虽然老派系统设计者会因为追求系统的完整性而牺牲掉简单性，Unix开发人员却将简单性提升到最高优先级。因此，老派系统设计者常常会抨击Unix的设计理念，这并不是因为他们否定这种做法，而是他们觉得这有点儿本末倒置。在这个意义上，Unix要逊于那些正常系统。

另一方面，Unix的狂热爱好者会指出糟糕的特性反而拥有长久的生命力，并坚称糟糕的反而效果更好。他们会说，看看VHS[①]录像带格式吧。相比索尼的Beta录像带，VHS录像不仅大而笨重，录音效果还很糟糕，它们远比不上光盘。然而与Beta格式相比，很明显是VHS录像带主导了着家庭视频市场。当然，今天的DVD早已取代了VHS录像带，因为它们更价廉物美。

同样，PC机的Windows用户界面也远远不及苹果机上那几乎完美的用户界面。而且即使从用户角度出发，PC机也比苹果Macintosh要糟糕得多，但PC机的占有率

① VHS是Video Home System的缩写，意为家用录像系统。它是由日本JVC公司在1976年开发的一种家用录像机录制和播放标准。——编者注

还是超过了苹果机。

Unix成功的原因之一就是，作为操作系统它在许多方面的感觉都比不上其他系统。人们通常不采用Unix来执行所谓的严肃工作，这些任务通常都会留给企业级商业操作系统。Unix一般只在那些硬件配置较低的机器上工作。它被广泛地应用于小型机，这些机器虽然缺乏大型机那样的强劲性能，却足以胜任日常事务的处理。由于小型机只是用来处理更次要的任务，因此，至少在科学界关注它之前，硬件供应商也就不会愿意对这个小型机操作系统投入大量资金。这种趋势在工作站出现之后日益明显。移植一个便宜买来的系统变得越来越经济实惠。

今天，我们不难找到比Linux上的免费软件更加好用的商业版本。大多数免费软件并不像商业软件那样，具备强大的功能和华丽的界面。然而，免费软件应用的发展态势却轻而易举地超过了商业软件。

现在，一些厂商和业内组织正在努力使Linux表现得更好。他们希望能够借此甩掉它“更糟的反而效果更好”的特性，并最终能受到人们重视。这可是一个致命错误。因为，如果Linux真的成为一款在各方面表现俱佳的系统，那它就面临灭绝的危险。想要“文武双全“？它就需要牺牲简单性来实现完整性。一旦出现这种情况，一款完美体现Unix哲学准则的新操作系统就可能会取而代之，强过这个进化版Linux系统。

7.10　层次化思考

几年前，我第一次觉得有必要给我的女儿解释该如何采用目录层次结构来组织文件。在此之前，她只了解一些文件系统布局的基础知识。她知道家中电脑的硬盘上存在着一个以她名字命名的目录。如果她在图形用户界面中使用鼠标指针点击正确图标，她便能够获取自己的文件清单。

年轻人虽然没什么经验，却有的是时间。作为一个9岁的孩子，她有足够闲暇来收集图片，她收藏的图片甚至多到屏幕无法完全显示文件清单。为了让她能更好地管理自己的目录，我给她演示了创建多个子目录来整理图片文件的好处。没过多久，她便意识到自己也可以在目录底下创建新目录，最后她将文件夹打理得井井有条，嵌套目录甚至深达5层。

她学会了层次化思考。

这只是一个简单的想法，但就像Unix上的其他众多事物一样，它有着深远的意义。虽然对于今天很多人来说，文件系统的分层次布局似乎是件不言而喻的事情，可在过去却不是这样。早期的操作系统往往会将系统相关的文件放在同个目录里面，而且在目录树中与其他用户文件夹位于同一级。在过去的日子里，人们并没有发现层次思考的好处。

大多数现代操作系统都采用层次结构来组织文件和目录，Unix也不例外。然而，Unix与其他系统有着细小区别，也就是它的语法，人们可以引用深嵌于多个目录级别中的文件。Unix中“/”字符使用起来很方便，它能够分隔文件路径名的组成部分。因此：

```
/usr/a/users/gancarz/.profile
```

代表着gancarz目录下的.profile文件。这个gancarz目录位于users目录下，顺此类推可以一直回溯到“/”目录，也就是根目录[①]。Unix文件系统的层次结构本质上是一棵倒置的树，根目录位于所有连续分支（目录）和叶节点（文件）之上。

Unix也采用了层次结构来组织其他元件。例如，Unix上运行的任务，也即进程同样采用了树结构。1号进程是init程序，作为树的根。所有其他进程（包括用户会话）都是init的后代或子进程。进程通过创建自身副本来繁衍子进程，并将自己标记为该子进程的父进程。如同现实生活中一样，子进程会继承父进程的属性。

另一个可以用来说明Unix层次思想的范例是X Window System的用户接口库。其中的资源管理器支持用户界面对象（如按钮和菜单）去继承其他层次用户界面对象的属性。这个强大思想使得我们可以为用户应用程序中的不同组件定制不同字体、颜色和其他属性。

除了Unix系统上的实际应用之外，层次化思考还有一个哲学上的原因。没有人能否认大自然也是一个分层组织的结构。让我们用一句陈词滥调来说明吧，从一个橡树种子成长起来的大橡树最终会繁衍出种子；而这些新种子又会繁衍出更多的橡

① 微软的Windows操作系统也采用层次结构来组织文件及文件夹。但是，用户必须知道文件位于哪个物理驱动器或是分区（例如，C：盘）。而Unix上的根目录却代表着文件树的顶端，它包括了系统上的每一个驱动器，而不是每个单独的物理驱动器对应各自的根目录。Unix用户不必去考虑系统上的物理驱动器和分区布局（这通常是非常复杂的）。在Unix下，所有目录和文件组成的是一棵大树。

树。自盘古开天以来，生命周期就是这样的生生不息。同样，在人类世界中，父母生出的孩子长大后也会繁衍子女。我们可以说，家谱这一想法的根源借用的正是森林中树木的概念。考虑到Unix是模拟大自然而构成的分层结构，这恰恰标志着分层思考的确是个好想法。

在这一章中，我们已经探讨了Unix哲学的10条小准则。我并不期望所有Unix或Linux社区的成员都会同意我所讨论的一切观点。我并不在意。Unix社区就像这个自由世界的其他地方一样，允许个人的自由表达，有时候甚至还能容纳截然相反的观点。

在其他操作系统上，大家也可以找到Unix包含的一些想法。我们无从知晓Unix到底是这些概念的开山鼻祖还是受益者。然而，许多Unix用户和程序员都有意无意地遵循着这些理念，Linux社区也一样。

第8章 让Unix只做好一件事

迄今为止，我们探讨过的大部分内容都是抽象的概念。虽然我们已经看到每一条Unix哲学的准则背后都有着现实的原因，但有些人仍然坚信这些准则并不适用于现实世界。“小即是美”听起来确实不错，可那些大规模的任务怎么办呢？是否真的能够采用小程序的集合来构建大型完整应用？没有CUI的程序有实际用处吗？可以肯定，这都是一些很实际的问题。你会在本章找到它们的答案以及其他相关内容。

让我们来研究一下MH，一个由兰德公司（RAND Corporation）开发的电子邮件处理应用系统。它由一系列小程序组合而成，带给用户强大的电子邮件操纵能力。从这个复杂的应用程序可以看出，使用小型组件来建立大型应用程序不仅是可行的，在设计理念上甚至更胜一筹。

如果某程序很有价值的话，人们就会把它从一个平台移植到另一个平台，MH应用就是这样。它一直保持着最初的基本想法，且还在年复一年地发布改进版本。在今天的Linux发行版上，新MH（即NMH）程序已基本取代了旧MH。NMH整合了所有MH的主要特点，而且最重要的是，它还坚持MH最初的理念。出于讨论的目的，我们在这里将探讨MH背后的思想，但这些思想对NMH同样适用。

在基于Web的电子邮件出现之前，多年来Unix环境中的电子邮件几乎全部是由两个程序（/bin/mail和Berkeley Mail）进行处理的。尽管Berkeley Mail的使用非常广泛，可其实这两个程序算是不怎么遵循Unix哲学理念的反面例子。它们都采用了CUI，而且不太具备过滤器的功能，也都不是小程序。它们均没有专注于做好一件事情，而是在单一的用户界面下纳入一系列与邮件处理相关的功能。

MH提供的功能相当于/bin/mail和Berkeley Mail这两个程序的总和。它集合了一

系列小程序，每一个都能执行这两个邮件程序中的某单一功能。MH还提供了其他一些这两个邮件应用程序没能提供的小程序。它仿佛在强调一个事实，在这个以Unix哲学理念为蓝本的应用程序上，人们可以轻而易举地添加新的功能。

表8-1是MH应用所包含命令的部分清单。

表8-1　MH应用所包含命令的部分清单

命　令	描　述
ali	显示邮件别名列表
anno	注释邮件消息
burst	把消息分割成多个新消息
comp	编写新邮件消息
dist	将消息重新发送到其他地址
folder	设置或列出当前的文件夹或消息
folders	列出所有的文件夹
forw	转发消息
inc	包含新的邮件
mark	标记消息
mhl	产生MH消息的格式化列表
mhmail	发送或阅读邮件
mhook	给MH接收到的邮件加入hook功能
mhpath	显示MH消息和文件夹的全路径
msgchk	选择消息
msh	MH shell命令（也是BBoard 阅读器）
next	显示下一条消息
packf	将整个文件夹压缩成一个文件
pick	通过内容来选择邮件消息
prev	显示前一条消息
prompter	启动提示符编辑器
rcvstore	异步添加新邮件
refile	将消息归入其他文件夹
repl	回复消息
rmf	删除文件夹
rmm	删除消息
scan	产生消息逐行显示列表
send	发送消息
show	显示（列出）消息
sortm	排列消息
vmh	MH的可视化前端

（续）

命　令	描　述
whatnow	为发送消息启动提示界面
whom	报告消息发送的对象

MH使用一组文件夹来组织用户的邮件。每条邮件消息都会存储为各个文件夹中的独立文件。此外，收件箱（inbox）文件夹还有一个特殊用途：一开始inc命令会把用户系统邮箱中的内容放置在收件箱中，一旦这些邮件进入收件箱，就可以选中它并执行其他MH命令，并且不需要去指明它所在的文件夹。

从上面的列表你可以看出，MH提供了能够从全面的电子邮件应用程序中获得的所有功能。scan命令可以显示当前文件夹中所有邮件的发件人、日期和邮件主题行。show、next和prev命令分别显示所选择的邮件内容、当前文件夹的下一封邮件和前一封邮件。comp和repl命令可以创建一封新邮件或回复现有邮件。

MH和其他Unix邮件程序的最主要区别就是，你可以从shell提示符这一级来调用任何一个MH邮件处理程序。这使得MH具有了极大的灵活性。由于每个功能本身就是一个小命令，因此我们可以像调用任何其他Unix命令一样去调用它们。

这些命令还可以当作过滤器，也能为shell脚本所用。比如，scan命令的输出结果可以通过管道发送到Unix的文本搜索命令grep，从而让我们迅速查找到特定发件人的邮件。如果文件夹里的邮件太多以至于屏幕显示不下scan的输出结果，还可以通过管道将输出结果发送到Unix的more命令，让用户一次查看一页列表。如果用户只想看一看最后5个结果，那采用小技巧scan | tail -5便能达到效果[①]。

其实，关键就在于用户能够灵活机动地操作邮件，而不受制于最初程序员为MH开发的功能。只需要将MH的命令与其他Unix命令相结合，就能创建出一个有用的新功能。不需要开发新的程序，用户就可以根据自己的需求将各种命令混搭在一起。如果现成的命令无法满足需要，你完全可以不费什么大力气便创建好新的功能。基于这种简单却强大的架构，新功能整合完毕后就能立刻在用户命令行中使用。

关于MH人们经常提出的一个问题就是：怎样才能记住有哪些命令？例如，那

① 列举的方法并不是执行这些功能的唯一方法。如同Unix的其他应用一样，条条大路通罗马。这里展示的例子只是用来说明Unix风格架构的灵活性。

些用惯了Berkeley Mail的人们知道总是可以键入“?”来获取可用命令列表。在这个方面，MH表现得相当不错。它虽然没有CUI，但你还是可以随时参考在线的Unix帮助手册。这些手册页面的详细程度大大超过了Mail命令提供的简短说明清单。MH的操作形式也因此大大改善，因为通常每个单独的命令都对应着整页（或更长）说明文档。

可那些对CUI情有独钟的用户该怎么办呢？别担心，MH为这些喜欢每执行一个步骤就得到相应提示信息的用户提供了一个msh命令。不过很显然，只要不特意这么做，用户是不会被锁定在msh命令中的。用户可以自由地编写运行在MH命令集上层的用户界面。例如，xmh就是其中一种=，它作为客户端运行在X Window System上。xmh应用中的按钮和菜单项只是调用底层相应的命令。不过，xmh呈现给用户一个无缝接口，看起来它好像就在直接执行邮件处理程序中的应用。

MH提供了如何在Unix下建立复杂应用程序的绝佳范例。图8-1显示了一种常用做法。

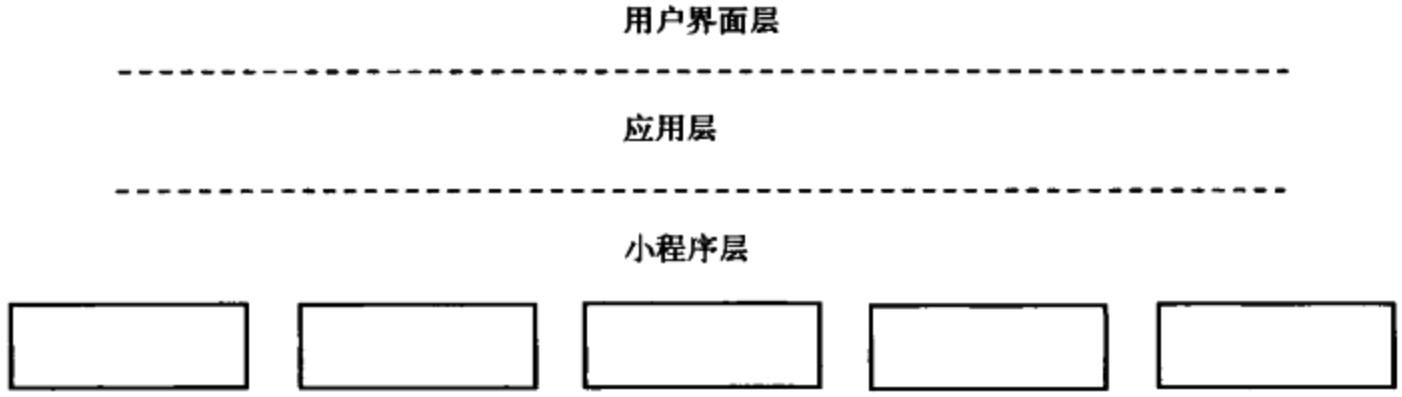


图8-1　常见的应用程序设计结构

该应用的小程序层由一组Unix命令或shell脚本构成，每条命令能执行一个功能。当新的功能需求出现时，人们可以轻而易举地创建其他小程序以提供所需功能。另一方面，如果只想要此应用的一个功能子集，也不需要额外再做什么工作。应用层可以不去调用那些用不着的程序。

应用层决定着哪些程序是用户的必备功能。这些功能本身是由小程序层来执行的，这一层封装了业务规则，并给底层应用提供边界条件。应用层其实就相当于小程序层和用户界面层之间的联系纽带。通过在单一环境中整合小程序层的基本元素去建立起小程序之间的关系，并提供将它们联系在一起的框架。换句话说，它就是一个集成层。

在调用应用程序的时候，用户在屏幕上看到的只是用户界面层这一部分。应用程序具有高度可塑性，它能够根据展示效果的需要来提供各种风格的用户界面。在Unix环境中有三种最常见的用户界面风格：命令驱动的用户界面，也就是从命令行来传递参数；滚动菜单式界面，这是通过一系列与用户的交互式对话来收集参数的CUI；图形用户界面，通常出现在有图形功能的PC上，比如X Window System。

经过良好构建的应用层允许用户自行挑选不同风格的用户界面，这就意味着用户界面层不能通过硬性编码方式来实现。用户应当根据具体环境状况来选择适合自己的用户界面。因此，如果只是偶尔使用，用户可以借助图形用户界面提供的便利性；但如果性能因素很关键，就可以选择可滚动的菜单界面。MH和它的辅助程序xmh就是应用程序集成灵活性的绝佳范例。

Unix 哲学：综述

在前面的章节中，我们了解到整体的效果往往要大于各部分的总和。对于Unix哲学本身来说更是如此：几乎每一条准则都无法独立存在。使用纯文本文件来存储数据听起来好像没什么道理，除非有相应工具来操纵这些文件。如果没有shell脚本和C语言这些实现手段，舍效率而取可移植性似乎略显单薄。假如小程序不能作为过滤器来使用，那编写小程序的集合也就失去了意义。

Unix哲学就像大型游乐园里的水上滑梯。你只能顺着弯道往下滑，而不能随意跳过其中的弯道——这么做你只会摔到地上弄个头破血流的下场。许多人早就发现，如果只是东一条西一条地采用Unix准则，那Unix哲学便达不到理想效果。

因此，只有将这些Unix准则集成在一起使用，Unix哲学才会显示出它博大精深、强大无比的一面，它们相辅相成。如果有人挑出单条准则来抨击，我们完全可以用其他准则蕴含的道理作为回应。有一句老话“合则立，分则垮”非常形象地说明了其中的道理。

让我们来重温一下Unix哲学。不过这一次，我们将侧重于阐明各个准则之间的关系，这主要是想让你了解只有整体运用这些准则才能发挥协同效应。

小程序有绝对优势。它们易于理解，相比大玩意，人们与小东西打交道更为方便。易于理解也就意味着它们更容易维护。因此，最终我们反而会获得更高的成本

效益。小程序占用的系统资源较少。它们能够更快载入、运行然后释放，从而产出更高的效率，而高效率通常会受到可移植性的影响。最后，小程序可以方便地与其他工具结合在一起。如果只有这些小程序，那它们的力量微不足道。然而，它们却可以和其他小程序密切配合，使程序员和用户能够迅速构建新应用。

小程序应该保持它的专注力（即，应该只做好一件事）。在单个程序中，解决一个问题比处理很多问题更重要。我们可以将大型复杂问题分割成各个小部分，这样就可以一次处理一点。有了这些只做好一件事的小程序，我们就能毫不费力地在其他应用中重复使用它们。小程序的功能定义非常明确，不会被细枝末节而模糊了程序本身的功能定义。

只做好一件事的小程序不会成为复杂的庞然大物。那样的应用程序往往包含很多杂乱无章的代码，而且难以和其他应用程序集成在一起。小程序清楚地了解，未来终有一天，今天的功能看起来会是那么的不完善，或更糟糕，变得过时了。尽管大型的单一化程序能够涵盖到当前已知的所有情况，小程序却深刻懂得软件一直在进化的道理。是的，软件从来都是完不成的，它只是会被发布出来。

世界上的软件一直在不断进化，每个人都需要持之以恒的学习。没有人能够绝对准确地预测未来软件世界的发展方向。我们能做到的就是抱着它在未来一定会发生变化的信念，去构建满足当前需求的软件产品。因此，与其花费数周或数月时间撰写设计规格文件，开发人员不如就着计划写一份概要性的简单文档，然后朝着设定好的方向努力。

尽快建立原型是一个重要步骤。它可以帮助设计者快速构建供他人使用的应用程序，以便尽早地朝着构建“第三个系统”的方向前进。建立原型能够加速通向未来。它们鼓励你在事情可控的状态下随机应变，而不是等到一切定型了才采取举动。它们会让你知道哪些做法是可行的，当然最重要的是，哪些做法是不可行的。

采用那些只做好一件事的小程序来逐步构筑原型，事情就会容易得多。这样只需付出最小的努力就能添加新的功能。

要记住，软件永远是完不成的，它会一直发展下去。在软件成长的过程中，如果它被移植到新的硬件平台，那无疑是变得更有价值。当新架构出现的时候，可移植的软件便可以迅速地利用上它们的优势。因此，在构建原型的时候，我们应该舍高效率而取可移植性。只有可移植的程序才能生存，其他一切很快就会变

成过眼云烟。

可移植的数据应该和可移植的应用齐头并进。只要有可能，请尽量使用纯文本文件来存储数据。再强调一下，我们无法预知未来，所以你不知道数据的最终发展方向。你大可不必担心可移植性数据的效率问题。新的硬件平台将大大加快移动数据的速度，甚至会比目前最快的机器还要快。请记住，可移植数据也简化了达成“第三个系统”的过程，因为在所有阶段里，这些数据都是人类可阅读的。

可移植的软件在编写时就可以从预料不到的硬件平台上找到自己一方天地。从某种意义上来说，通过编写可移植的软件，你就为计算技术出现以来的软件宝库作出了自己的贡献。请记住不管为这个世界贡献了什么样的软件，你都有权利去获得其他软件。当你的软件被应用于各种平台时，它都将增强用户的能力，帮助他们去完成任务。

所有你付出的努力会像滚雪球一样将效果累积起来，其他程序员完成的工作也是如此。如果你想利用软件的杠杠优势，就要绝对避开NIH综合症。你可以创建新的应用，但不要浪费时间去重复创造别人已经写好的软件。世界上的现有软件其实就是一大笔财富。当然，你在计算机领域播撒下种子准备收获成果的时候，还请记住要合法获取资源。

为了尽可能拓展软件的影响力，请尽可能采用脚本和其他高级抽象工具。它们能够充分利用他人早已完成的工作。脚本可以放大应用世界里其他人的工作成就，使你达到事半功倍的效果。

如果有一大堆小程序可供使用，那脚本的构造就会更为简单。不过，假如这些程序还是需要用户直接输入参数或数据，那它们的实用性就会大大降低。因此，我们应当避免使用CUI。相反，我们应该将所有程序都视为过滤器。程序本身并不创建数据，它们只修改数据。尽量让程序变得简单，从而能在其他的地方使用。

协作。是的，我们讨论的正是协同效应。

就是这些吗？嗯，是的。大多数Linux和Unix程序本身并没有什么特别值得称道的长处。对于任何Unix上的解决方案来说，你总是能在其他地方找到更好的文字处理程序、更别致的电子邮件应用、更为清晰漂亮的文件夹视图。（当然，Linux上的一些图形用户界面应用也令人印象深刻。）但是，你可以尝试一下那些臃肿的文

字处理程序做不了的事情[①]，由此你会发现，Unix的功能确实强大无比。

其他操作系统的支持者会声称某个特定操作系统的表现要比Unix出色，某些情况下可能属实。即便如此，Unix系统（以及后续发展成的Linux）很有可能在短时间内提供一个90%的解决方案，而且花费的时间只相当于编写某个全新应用程序所需要的一小部分时间。此外，Unix可以为几乎所有情况提供这90%的解决方案，即使是在那些看似不可能的状况下。

不过，我们就此打住。不要让人觉得我们是在自吹自擂。如果你想了解Unix哲学与其他操作系统的比较状况，请继续阅读下一章节。

① 如果你曾试图根据动态变化的业务规则来修改一千个文件中的文本字符串，那就能体会到文字处理程序的局限性。这类繁琐的工作需要一款非常好的过滤器程序。另一方面，如果你想创建文档的初稿，那么文字处理程序算是个不错的工具。

第9章

Unix和其他操作系统的哲学

至此，我们该简要讨论一下Unix之外其他几个操作系统的哲学。我们并不打算做过分深入的研究，也不会就比较各个操作系统的哲学而展开长篇大论。但是，粗略地研究一下软件世界中的许多其他“门派”对我们还是会有启发的，至少能够说明为什么Unix的做法与其他软件设计理论是如此大相径庭。

本章将着眼于三个历史性操作系统的设计理念。每一个系统都在各自的领域取得了成功，并发展了一批自己的忠实信徒。每个操作系统都与Unix和Linux存在着一些共性，同时又具备一些独有的特色，甚至是一些截然不同的特性。

出于种种原因，这些操作系统不仅受到硬件环境的限制，而且是针对具体的硬件平台而开发的。想让用户为这些相对不太灵活的系统真金白银地掏出几十万到几百万美元，供应商需要很强大的营销能力。当时，在折扣零售店和跳蚤市场并没有数千个应用程序可供挑选。个人电脑并不便宜，品种也不是那么丰富。没什么人会在自己家地下室的旧物堆上闲置个几台落满灰尘的旧电脑。

更新更快的硬件最终还是问世了，加速了这些老旧操作系统被淘汰的过程。它们不具备Unix的可移植性，也跟不上硬件设计者推出新机器的节奏，新机器的存储空间在不断增加，还有专用外设和功能先进的图形设备。唯一的例外就是微软的DOS操作系统（MS-DOS）。然而， MS-DOS并不是一款可移植的操作系统。虽然开发人员逐步将它移植到了更强大的机器里，但基本上这些机器采用的都是相同的架构。在较新版的Windows操作系统中，微软已经完全移除了MS-DOS，转向Windows API和图形用户界面。

尽管本章讨论的操作系统基本都已退出历史舞台，但今天，它们的设计理念

仍然以某种形式存在着。例如，虽然雅达利家庭电脑系列很久之前就已停产了，可它的操作系统中应用的人体工程学概念在当今最热门的游戏机中仍在继续实现和优化。

假如我曲解了以下操作系统设计者的原意，在这里先表示歉意。正如你所怀疑的，记录总结操作系统的理念非常难。我只希望在某些方面抓住了设计师的本质目的。不过整体上，对大家来说，绝对的准确性不如总体的设计思路重要。因此，下面的讨论应该能够给我们带来一些有益思考。

9.1　雅达利家用电脑：人体工程的艺术

20世纪80年代，雅达利800和较为便宜的同系列产品雅达利400占据了家用电脑市场中相当可观的份额。最初它们主要面向游戏爱好者，后来却因为先进的图像和声音处理能力而声名鹊起。“星际奇兵”（Star Raiders）是一款模拟星际迷航（Star Trek）的游戏，它的3D空间游戏里有许多星星和呼啸而过的光子鱼雷，正是这款游戏帮助雅达利电脑占据了家用市场的强势地位，直到它被广受欢迎的IBM PC和物美价廉的Commodore 64取代。在今天看来，它的“玩家导弹”图形（精灵）和显示处理器是相当原始的技术。

Unix和雅达利家庭电脑操作系统的设计者拥有一个共同目标，他们都希望能够构建一个适合玩游戏的系统——也就是针对Unix上的“太空旅行”（Space Travel）游戏和雅达利机器中的多款游戏。这可能是他们赖以成功原因之一。人们通常会为那些给他们带来快乐的事情而努力工作。今天，很多开发人员编写Linux软件就纯粹是出于兴趣。对他们来说，这是娱乐，也是一种“极客”（geek）生活方式。除了满足生存和社交的需要，人们还有与生俱来的娱乐需求。对于计算机世界以外的人们来说，这看起来也许有点儿匪夷所思，但许多Linux极客确实把软件开发当做是一种非常有趣的消遣。

Chris Crawford对雅达利家庭电脑操作系统的设计产生了深远影响，当时他是这个设计小组中的一名成员。他的应用软件（其实也就是游戏）为后续计算机软件重新设定了一个评判标准。在他撰写的*De Re Atari: A Guide to Effective Programming*一书的附录B中，Crawford详细记录下了他的设计理念。虽然该部分主要焦点都是集中在人体工程学上，它同时也揭示了Crawford关于计算技术的许多总体设计思路。

Crawford认为计算机是一个缺乏外在物理特征的智能个体，它的思维过程是“直接的、分析性的和具体的”。他把这些特性与人类“联想性、集成性和发散性”思维模式进行了对比。正是这些思维过程的差异造成了人类和这些小型智能“侏儒”（Crawford给计算机起的名字）之间沟通交流的障碍。他坚信，程序员的目标并不是为了让这些“侏儒”变得更强大，而是要去消除“侏儒”与人类互动沟通之间的障碍。

这也正是Crawford的做法与Unix哲学的不同之处。Crawford强调软件最重要的一个方面是如何与人进行沟通，而Unix程序的最高优先级却是该如何与其他程序互动。虽然Unix程序最终也需要与人交互，但这不过是更次要的方面。

在用户界面设计采用的方法上，雅达利家用电脑和Unix操作系统大相径庭。根据Crawford的说法，雅达利机器上的软件力求封闭性，或是尽量减少用户的功能选项。

> 理想的方案就像一条“穿过坚硬岩石的隧道”。能够通向成功的道路只有一条。用户只能成功地完成任务，并没有其他的选择。[①]

想要创建最适合雅达利家庭电脑的程序，开发人员就必须限制选项的数量。程序只能给用户提供一种执行某个任务的方法，而且必须显而易见。它几乎不会给用户留出犯错空间，即使这意味着只能提供很有限的选择权。按照Crawford的思路来看，Linux和Unix世界的灵活性与适应性只会导致极大混乱。

现代游戏机的实现方法与雅达利家用电脑的理念相同。它们限制了选项的数量，使得孩子们不太可能作出错误选择，并确保玩家与计算机之间通常会有愉快却缺乏灵活性的体验。

限制计算机系统的选项是很有挑战性的任务。因为，用户在不断寻求最新快感，哪怕是在一个受限制的世界中，他们也想去突破窠臼。他们想冒险，但又不愿意太激进。他们希望能够时不时偏离正常轨道，同时又希望迷途知返的时候也能够安全回归。

雅达利的做法就好像他们认为如果给每个人都发把枪，那很可能会有人射中自己的膝盖。相比之下，Unix系统的做法却像是给门外汉用户发了一把冲锋枪，并装

① *De Re Atari: A Guide to Effective Programming*，1982年雅达利公司版权所有。

上20发子弹对准了他们自己。“膝盖中箭”的人是走不远的。

不出你的预料，Unix方式可能会造成破坏性的影响。Unix环境中的选择非常丰富，几乎没有什么限制。通常情况下用户至少有十几种方法来执行任何一项任务。这种极大的自由性非常考验用户，往往会消磨殆尽他们的耐心。如果选错命令的选项就会破坏数据，让他们无所适从，不知道该如何挽救。不过，最终大多数Unix高级用户会发现他们开始理解整个环境，整体思路也变得清晰。人们并不喜欢那些人为强加的限制，他们更加青睐程序的灵活性和适应性。此外，对于那些不想成为Unix高水平用户的人们来说，各种Linux发行版都附带有图形用户界面，从而隐藏起程序的复杂性。

主流电脑用户已经逐渐适应了这种选项日益减少的趋势。PC机上曾经有好几种办公应用套件备选，且各有所长。然而，今天微软的Office软件完全占领了这个市场。许多企业都拒绝使用其他的办公应用程序。有人辩解说，如果我们都采用相同的应用程序套件来创建文件，便能简化我们的办公事务，每个人都可以更轻而易举地共享文件。但它有一个缺点，这限制了人们的创造性。如果我们只是创造一种公开的文件格式标准，，那符合标准且有趣的新应用程序就会出现。除掉单一供应商提供的标准功能外，它们还会具备其他令人意想不到的功能。虽然微软Office应用的确很不错，是一种功能丰富的应用套件，但人们相信全球的创造力应该不只出现在微软公司总部雷德蒙德的范围之内。

我们只需要观察一下有线电视市场就会明白，消费者面临的选择非但没有减少，反而在增加。大约40年前，也就是工业时代的末期，美国用户只有三个备选的电视网络。而今天，普通的数字有线电视系统就能提供数百个频道。随着人口的增长，以及人们观看电视的兴趣变得更加多元化，这个数字还在只增不减。

信息时代，用户拥有更多的选择。“穿过坚硬岩石的一条隧道”变成了一系列为特定用户定制的通道。就像雅达利家庭电脑的用户一样，计算机新用户或是偶然使用的用户需要一种只有很少功能选项的用户界面，这种界面不太可能会出现错误情况；中级用户需要更多的功能和选项，但又不希望这会让简单的任务变得难以操作。没有人愿意一次性接收过量信息。那些高级用户则希望有权使用系统提供的所有功能，他们不希望系统的选项被用户界面隐藏。

这正是Linux得以大展身手的领域。借助诸如KDE这样的图形用户界面和其他环境，它就能够给新手用户提供多个入口点。随着桌面环境的改善，微软的Windows

和其他基于图形用户界面的操作系统将会失去它们对于新用户所拥有的领先地位，而慢慢屈从于其他桌面系统的冲击，其中一些系统可能会更易于用户使用。Linux下的高级用户也会发现它别具一格的开放式架构和接口给用户留出了大量弹性空间，这可是封闭式架构无法提供的。

现在，如果我们找得到Chris Crawford，让他重新为Linux编写一款超酷的“星际奇兵”游戏，该会有多棒。

9.2 MS-DOS：七千多万用户的选择不会错

在微软的Windows NT和后续几代Windows操作系统问世之前，MS-DOS是当时PC机世界中最主流的操作系统。曾经有一段时间，每天有七千多万人在使用MS-DOS。如果说操作系统的用户数量是衡量成功与否的唯一标准，那么MS-DOS可以轻松地声称自己是历史上最成功的非Windows操作系统[①]。

这种“羊群效应”（herd mentality）可以说是MS-DOS在美国取得巨大成功的决定性因素。20世纪80年代小型车让美国人慷慨解囊，而小型计算机也获得了数百万人的青睐，他们需要在个人用户级别上（即，几乎每一个人）存储和管理信息。在IBM构建了一款功能强大的机器并动用全部营销力量将其炒作成公众对“个人电脑”的成熟需求时，人群蜂拥而上地购买这款机器。是的，还有哪个公司能媲美如通用汽车这般对商用机工业革命产生的推动性力量？

所以，MS-DOS被公众广泛接受并不是因为它代表了操作系统设计的一大进步，而是因为它运行在一款被公众认为是更为安全可靠的系统之上。大家都说，“买IBM机器就是拿铁饭碗”（no one was ever fired for buying IBM）。将微软的MS-DOS雄心勃勃的营销策略与IBM PC“克隆”制造商相结合，你会得出一个结论，人们一般倾向于采用一种更为安全稳妥的方案。

随着时间的推移，这种现象开始进入良性循环。越来越多人购买并安装了MS-DOS系统的机器，应用软件的供应商也开始将MS-DOS视为首选平台。因此，更多MS-DOS应用程序出现在市场上，又使得MS-DOS吸引到更多的人。

① 另一种衡量成功的标准是看一个程序有多少非人类用户。执行某程序的其他程序越多，就越说明它就是更好的方案。参见第10章。

今天，作为Unix阵营中的新成员，Linux也经历了一个类似的周期。Linux上的应用程序越多，选择它的人就越多。

那么，MS-DOS背后的哲学到底是什么呢？首先，简单是最重要的。如果想为公众设计一款操作系统，那么对于目标群体的大部分成员而言，它要易于使用。因此，MS-DOS使用了简洁有限的命令语言。虽然对老手来说，它几乎没有什么回旋余地。但另一方面，详细的帮助和错误消息多少也抵消了一些该命令集限制性设置产生的负面效应。例如，以下就是MS-DOS中典型的通知消息：

WARNING! ALL DATA ON NON-REMOVABLE DISK DRIVE C: WILL BE LOST...PROCEED WITH FORMAT (Y/N)?

其次，在限制用户输入的同时增加系统的输出信息，MS-DOS的设计师使得用户更像是一名乘客，而不是司机。这有助于避免用户产生错觉误以为自己是坐在一台巨大主机的控制台前。不幸的是，这却违背了人们使用个人电脑的宗旨。

相比之下，Unix有一个相当强大的命令语言集，以至于那些高水平的用户有时候也无法充分利用它的全部功能。虽然它的命令语言看起来似乎有无穷无尽的选择，但是它的错误消息却非常简洁，令人头痛不已。比如，mkfs是一个具有潜在破坏性、用来初始化新文件系统的命令，在使用它的时候，所能得到的最强烈警告消息也不过就是——“Do you know what you’re doing？”（你知道自己在做什么吗？）

也许，MS-DOS环境下最令人惊讶的地方就是它包含了一些Unix的概念。例如，MS-DOS提供了一种类似于Unix管道机制的管道功能，以及一个树形目录结构。它还包含一条MORE命令，它实现的功能与Unix发行版中的more命令相差无几。这或许表明一些熟悉Unix的人士参与了MS-DOS早期设计阶段的工作。

Unix的管道机制可以在不使用临时文件的情况下，直接将一个命令的输出作为另一个命令的输入。下面是一个简单例子：

```
find / | grep part
```

这行命令实际上包括了两个命令——find和grep。find命令输出一份系统上所有文件的清单。grep命令筛选出那些包含了part字符串的文件名。管道符号“|”是将find命令的输出数据作为grep命令的输入数据的管道。

find和grep命令同时被调用，这采用的是一种被称为“多任务”的计算风格。它将find命令产生的输出数据直接传递给grep命令。

然而，MS-DOS的管道与Unix管道的不同之处在于，MS-DOS并没有提供真正的多任务能力。不管命令行中输入了多少条命令，它一次只能执行一条。

20世纪90年代初期，有一度看起来MS-DOS的后续版本可能会包含更多的类Unix功能。例如，MS-DOS第6版中添加了CONFIG.SYS和AUTOEXEC.BAT文件来支持多重配置，显然它想提供更大的灵活性。多年以来，Unix允许每个用户按自己的喜好定制.profile文件，从而提供了多重配置能力。然而，很明显微软对Unix的做法并没有太大兴趣，最终它还是为Windows系统选择了一条以图形用户界面为基础的道路。

9.3 VMS 系统：Unix 的对立面

如果说在Windows出现之前，MS-DOS是个人电脑操作系统之王，那DEC公司的VMS系统[①]就可以说是小型机操作系统的巅峰之作。在Unix获得全世界的关注之前，没有任何其他小型机的操作系统能够赢得这么多忠诚的追随者。

VMS的成功在很大程度上要归功于DEC公司的VAX计算机系列产品，同样的软件可以丝毫不加修改地运行在跨度很大的各种系统上，从桌面机到整个房间那么大的实验室系统。这种在二进制级别上的兼容性使得人们能够比较容易地构建庞大的、完全集成的环境，其中的系统便可以采用统一风格进行交互。

DEC公司的系统构建方法一度让行业内的其他公司艳羡不已。DEC公司的VMS系统和VAX产品线在小型机领域取得的成就无人能出其右。其他公司如IBM和惠普也曾试图采用单一架构来统一它们的整个产品线，可最后还是失败了。英特尔公司在运行Windows系统的x86架构上取得的成就可以算是对DEC公司这种“一个架构对应一种系统”概念的肯定。

VMS系统是一款闭源的专有操作系统，这意味着它的开发工作是由一家公司独自完成的，并牢牢地保持着软件的所有权，由此得到该软件带来的一切利润。虽然

① DEC公司后来将VMS更名为OpenVMS，这只是一个营销手段，用来抵挡那些“开放”系统（如Unix）供应商对市场的冲击力。考虑到它曾有过的辉煌历史，将一个像VMS这样的封闭系统命名为“开放式系统”也许是一种终极讽刺。

这个系统是由DEC公司位于新罕布什尔州纳舒厄城的VMS工程组编写，可它实质上代表着DEC公司全体工程师在头脑里思考了多年的总体思路。它与DEC公司其他操作系统有很多相似点。比如，DEC公司较早操作系统RSX-11的用户对VMS的命令解释器可以说是驾轻就熟。就像DEC早期开发的操作系统一样，VMS依赖着贯穿于整条硬件产品线的通用性，并借此来加强和扩大它的用户基础。

对于构成VMS哲学的想法，VMS系统的开发人员有着自己强烈的主张。就像雅达利家庭电脑操作系统的开发人员一样，他们认为得将用户完全屏蔽在系统内各种变幻莫测的情况之外，但此外用户还应该明了系统存在这些不确定因素。如果说雅达利的做法是在“从坚实的岩石中钻出一条隧道”，那么VMS的做法就是在隧道中挂上明灯。

其次，假如说VMS环境中采取的是“大即是好”的策略，那么越大就越受欢迎。这导致人们更热衷于实现大型复杂的环境，比如DEC公司的多合一办公自动化产品，它将许多与其密切相关的功能都集成在单一的用户界面中。在基于个人电脑的办公系统开发之前，VMS系统雄踞这个领域，算是一款最强大的办公套件。

VMS哲学的第三个要素是VMS系统的思维定势对DEC客户产生的深远影响。人们购买VMS系统是因为需要完成某项确定的任务，而且对该项任务要求的信息技术一无所知或是不太了解。虽然今天常见的Linux用户会说：“我需要这个框”或是“给我那个应用”，但常见的VMS客户却会问，“我该怎样使用你的系统？”这其中的差别可能看起来并不明显，这却深深地影响了VMS软件工程师编写系统程序的思路。

VMS工程师开发出的应用往往具有丰富的功能，但是用户界面底下隐藏的细节却更多，他们根本没办法一览系统所拥有的奇妙复杂性。尽管该系统实际上拥有很多有趣的选项，但显示给用户的功能选项并不是太多。Linux和Unix应用程序的设计者通常都不会限定用户的操作。典型Unix系统的选项往往包括一切：好的、坏的和那些人们认为应该省略掉的选项。

在一定程度上，VMS的哲学隐含了一个基本信念，那就是用户害怕计算机。考虑到现代电脑的普及程度，这听起来有点怪怪的。事实上，还是有很多人觉得计算机是个令人望而生畏的玩意儿。如果可以，他们宁愿忽视该伟大技术带来的好处，许多人希望电脑会自动消失。

Linux/Unix系统长期以来的倡导者Jon“maddog”Hall常常会提出一个棘手的问题：我们的爸爸妈妈是如何应对现代技术的？他说，父母那一辈人都有微波炉，却只用上了其中59%的功能；他们还有一个CD播放器，但不知道什么是四倍采样（quad oversampling）技术；他们拥有一台录像机，却从来没有制定过拍摄计划；他们不知道遥控器上的所有按钮都是做什么的。总之，他们拒绝去了解那些不易于使用的技术。

爸爸和妈妈最讨厌Linux和Unix这种“走自己的路”的做法，虽然Unix的倡导者并不愿意承认这一点。在很早之前，VMS哲学的支持者就要应对这种情况。VMS开发人员想出的解决办法是让那些需要条条框框的人们更易于使用，而不是编写那些能让用户DIY的工具包。

除了用户界面之间的明显差异外，VMS系统与Unix还有很多不同。在许多方面，它们可以说是处于两个极端，我们甚至可以称VMS系统确实是Unix的对立面。例如，VMS系统通常只给用户提供单一化的解决路径，而Unix却往往会提供十几个甚至更多解决办法；VMS系统喜欢采用有着多个选项、规模宏大的单一化程序来满足众多用户的需求，但Unix更喜欢使用小程序，每一个都执行单一功能且只有为数不多的选项；VMS系统最初是采用汇编语言和BLISS-32编写的，这些编程语言都与底层的硬件结构高度相关，而Unix使用的却是更高级的C语言，并且可以移植到许多CPU架构。

然而，VMS和Unix最明显的一个区别就是，VMS系统是一个闭源专有系统，而Unix（不管它有多少种发行版）却始终如一保持着开源系统的特性，尽管许多厂商都试图夺取控制权将其变成闭源系统，从而占为己有。

最终，VMS和它这种封闭式系统开发的做法在20世纪90年代初期遭受了Unix和开源系统的猛烈冲击。这并不奇怪。DEC公司的专有操作系统在开源系统不懈努力下节节退败。在70年代中期，人们见证了DEC的RSTS和RSX系统逐渐被Berkeley Unix取代的情景。然后在80年代初期，DEC的TOPS-20系统也只能无可奈何地看着自己的地盘被AT&T公司的System V Unix蚕食。

在这一章里，我们将Unix和它的设计哲学与计算机历史上的几个其他系统作了比较。我们已经了解到其中一条操作系统理念，即操作系统的用户界面就像一条“穿过坚硬岩石的隧道”。我们也已经看到另外一个系统受惠于羊群效应而变得流行。

人们总是喜欢追赶时髦事物，因为跟着所有人朝同一个方向狂奔感觉不错。最后我们还研究了一款优化了“穿过坚硬岩石的隧道”理念的操作系统并最终形成了精密、单一化、具备工业特质的系统。

Unix和它的工具采用的做法经得起时间的考验，它们的存活历史会比所有这些操作系统都来得长久。也许是因为Unix哲学是一种前瞻性的操作系统（或软件）开发方法，它在不断地展望未来，它认为这个世界是在不断变化的。它也清楚即使我们了解现在的一切，但我们的知识体系仍然不完整。

每一款我们讨论的系统都曾经有过辉煌，在某些情况下也能够成功地给出可行的计算技术方案。但没有任何一个操作系统可以兼顾到最广泛用户层面的需求。雅达利家用电脑的操作系统限制了用户的功能选择。虽然因为羊群效应，MS-DOS系统一度获得了巨大成功，最终它却只能眼睁睁地看着自己被一款带有图形界面的操作系统取代，就因为它的功能还不够强大，其命令集也过于单薄。VMS系统虽然在一段时间内抢占了服务器市场，可在开源系统初露锋芒之际，却无法与之抗衡。

迄今为止，你一定会很奇怪，为什么没有见到关于微软Windows系统的讨论。我们为什么不把Linux和Unix与微软的Windows放在一起作比较？难道Windows不是那个除Linux和Unix之外唯一重要的操作系统吗？为什么要讨论那些现在几乎没有什么人使用且早已过时的操作系统呢？

我要指出一点，Windows赖以成功的一个原因就是它限制了新手用户的选择，就像雅达利家庭电脑的操作系统一样。华尔街很清楚微软早已经取代IBM公司成为大众的宠儿。每一天你都会听到人们对Windows的吹捧，夸它是一个“精密且具备企业级计算强度”水准的理想平台。

你能看出其中的相似性吗？不错。你跟上了我的思路。对那些还不甚了解的人们来说，请等上几年到你装好Linux的时候再说吧。在我们为隧道点亮明灯之后，你会更喜欢Linux。

第 10 章

拨开层层迷雾：Linux与Windows的比较

10

就算一群人认同你的观点，也不代表你就是对的。我同样可以找出另外一群认同我的观点的人。

——摘自与Don “Smokey” Wallace的对话

在前一章中，我们对Linux和Unix的哲学与其他操作系统的理念作了比较。虽然这些系统在各自领域里也算是举足轻重的角色，但它们对世界的影响远远比不上微软的Windows。不过，我们探讨这些系统也是有必要的。它们不仅表现出“专有系统”共有的特点，而且体现的设计理念在各个方面都与Unix哲学背道而驰。此外，这些设计理念的追随者理所当然地会发现有一个新的系统，也就是微软的Windows继续贯彻了这些系统的做法。

如果你仔细研究微软Windows背后的设计理念，就会发现它和VMS操作系统非常相似。这并非是偶然。它们本质上的相似性是因为Windows的设计工作是由David Cutler负责的，他曾是DEC公司负责VMS系统（以及VAX硬件）开发工作的首席工程师。多年以来，Cutler都是DEC公司众多操作系统的推手，VMS可算是他的代表作，或者看起来似乎是这样。20世纪90年代初期，在VMS系统到达顶峰状态后不久，Cutler却另谋高就去别处追寻他的梦想。他最终在微软找到了他的方向，并成为Windows NT设计工作的领头人，最终这个系统攫取的市场份额比VMS系统的还要大。

当时，很多人猜测Windows NT其实应该算是VMS系统的下一个版本。事实上也有人指出在字母表中“W-N-T”正是分别紧邻“V-M-S”的那三个字母，有传言

称Cutler是这么回应的："为什么这么久了你们才发现[1]？"

这并不意味着Cutler对微软Windows交互式图形用户界面有着深远的影响。在Cutler加入微软之前，微软早就走上了发展图形用户界面之路。不过，除掉图形用户界面之外，Windows在很多方面与VMS颇为相似。Windows是一个闭源的专有操作系统。尽管它的商业伙伴和竞争对手曾多次呼吁它开源，微软却从未将Windows操作系统的内部机制公开出来。就算迫于市场和法律的压力，恐怕微软也永远不会公布所有的Windows源代码。微软就像是将Windows的源代码视作为传世之宝，并牢牢把持着它的所有权。无论是对还是错，似乎执意认为如果将Windows源代码公开，它便会失去自己的竞争优势[2]。

Windows操作系统的创造者也奉行越大越好的软件设计方法。他们宁可发布越来越大且因此而消耗很多系统资源的单一程序，也不愿意采用有趣的方式将一系列小程序整合在一起来构建一个新的软件。

微软的Office套件恰恰就是一款这样的产品。它没有像Unix那样去使用一系列的小插件，然后根据需要来装载模块，它采用的是在启动时就加载一切的做法。这导致系统不断地需要更大容量内存。虽然内存芯片代工企业乐于见到这种系统资源饥渴症，可它却迫使用户在内存和CPU升级上过度增加开销，其实如果应用程序能够运行得更为高效的话，这些系统开销都是不必要的。相比之下，Linux操作系统因模块化结构而更为节约资源。虽然有了大量内存空间，很明显任何操作系统都能从中受益，但是就算是在那些只能满足部分Windows系统所需资源的机器上，Linux仍然能够有效地运行。

然而就设计思路而言，Windows最像VMS，它们都基于同样的设想，那就是用户害怕使用计算机。我们可以回想一下，VMS系统的开发人员想当然地认为用户视电脑为一个可怕的东西。在为Windows设计软件的时候，该系统的软件开发人员不遗余力地想让操作系统能够更易于新手使用。

这就是Windows哲学和Unix哲学之间的根本区别。我们在前面曾经讨论过，Unix系统从来没有试图去满足新手易于使用的需要。它的文本用户界面与许多灵活

① *The Computists' Communique*卷2第26号，1992年6月24日。

② 微软时不时地会与合作伙伴分享只读形式的Windows源代码。但微软几乎不允许任何其他公司修改Windows的代码。微软通常都是自己完成所有的改动。

方法仍然顽固地坚持着不那么容易让新用户快速上手的风格。用户必须刻苦学习，然后才能利用上它内在的强大功能。他们必须花费几天、几周甚至几个月才能成为Unix的熟练用户，从而去执行一些在Windows系统上花个几分钟就能掌握和执行的任务。

Linux企图通过提供诸如KDE和Gnome这样的桌面环境（图形用户界面）来弥补新手和专家用户之间的这种差异。KDE和Gnome的用户界面与Windows的类似，支持用户使用常用的“点击式”功能并添加了自己独有的一些新功能。这些都构成了一个保护层，使得用户无法接触到更多底层的命令，它往往也对那些处于起步阶段到中级水平之间的用户限制可用的选项。有经验的用户则可以通过命令行接口来接触到真正复杂的底层事物。

Linux拥有巨大的灵活性，它允许你采用双环境来操作机器，即基于文本的用户界面和图形用户界面并存。不过，它避开了一个重要的问题。既然Linux可以提供类似于Windows的环境，那我为什么不直接运行Windows来获取第一手的真切感受呢？如果你已经拥有了Windows，你为什么还会需要一个很像Windows的玩意？正如一首歌唱的那样“亲爱的，什么都比不上正品带给你的感受！”

一个答案就是，虽然Linux和Windows的桌面看起来也许很相似，但其实它们的架构是明显不同的，因为Windows的图形用户界面与底层的操作系统紧密集成在一起。Linux中的X Window System与Windows却有着本质的区别，它只不过是运行在操作系统上的一个应用程序，但Windows的图形用户界面和操作系统实际上却是一个完整的实体。它们的结合是如此地紧密，任何在图形用户界面中发生的错误都会对操作系统产生灾难性的后果。如果因为某些原因图形用户界面挂机了，用户完全没有其他可以取得系统控制权的方法。几乎所有用户和操作系统之间的互动都得通过图形用户界面来完成。除此之外，没有其他的方法能让用户去操作整个系统。我们可以对比一下Linux，人们完全可以采用不同的用户界面来取代X Window System。事实上，Linux的Gnome和KDE环境正是这所谓“不同的用户界面”，它们都位于同一个操作系统上的。

图形用户界面和操作系统之间的区别由来已久，其实它们就是一种形式和功能的关系。*功能*是指整个系统的（即用户所需要的）内容。应用程序的存在是因为我们需要功能。*形式*则代表着如何将功能呈现给观众。形式很重要，因为它通过赏心悦目的方式将信息传递给观众。但是，如果信息里面没有任何功能或内容，形式也

只不过是空洞无物的。对操作系统来说，华丽的图形用户界面没有任何意义，除非它能为用户提供所需要的功能。换句话说，内容为王。

10.1　内容为王，傻瓜

如果你出生于1950年之后，那你与计算机第一次真正意义上的亲密接触可能就是使用WIMP（Windows、Icons、Menus、Pointers，也就是窗口、图标、菜单、指针）风格的个人电脑。电脑总有图形用户界面，在里面你可以点击那些被称为“图标”的小图像。你可能认为，微软发明的窗口、菜单以及所有令人愉悦的视觉效果给个人电脑带来了很多乐趣。

实际上，WIMP风格的用户界面并不是微软的首创概念，施乐公司（Xerox）才是它的发明者，你可能以为后者只生产复印机吧。1981年施乐公司的帕洛阿尔托研究中心（Xerox's Palo Alto Research Center，Xerox PARC）推出了Xerox Star产品，它是第一款具有明显WIMP用户界面的产品。其他早期的系统如Smalltalk和Alto也体现了一些Xerox Star系统中的想法，不过，第一次将所有的元素组合在一起的还是Xerox Star系统的开发人员。

我们仍然可以从微软的Windows系统、苹果的Macintosh和各种Linux图形用户界面环境中找到Xerox Star系统具备的很多风格。它被比喻成是一个“桌面”，也就是邀请用户像对待办公桌上的物品一样对屏幕上的对象进行操作，这也是Xerox Star系统首创的概念。位图显示、图标和鼠标的使用都源自Xerox Star系统的用户界面。

同样，微软也没有发明万维网，Tim Berners-Lee才是它的发明人。他对网络的愿景并不是想成为坐拥大量信息的“沙皇”，而是真诚地希望能看到人们更好地沟通。1990年他和Robert Cailliau开始实现他的设想，他们在横跨法国和瑞士边境名为CERN的欧洲物理实验室里编写完成了一个简单朴实的网页浏览器。[①]

WIMP用户界面和万维网的存在主要是为了一个目的：提供内容。Xerox Star系统的位图显示模式使得我们能够显示图片、动画、视频和更具可读性的文字。有了网络，我们就可以将数据从功能强大的服务器移动到用户的机器并显示出来。将

① 请参考1997年5月19日出版的《时代周刊》中的文章The Man Who Invented the Web，想了解更多关于Web的历史，还可以参考网站 http://www.cern.ch。

这两项技术结合在一起就能够把丰富的内容传送到桌面了。

在首个网络超级搜索引擎AltaVista的初期阶段，开发人员喜欢说的一句话就是："内容为王，傻瓜!"他们这么说的意思是，无论你是否拥有最快的搜索引擎、最漂亮的图形界面或是最耀眼的桌面，这些都不重要。那些花哨的效果给用户留下的印象持续不了多久。最终，你必须提供内容资料以满足个人的需要。否则，用户就会去到别的地方。如果无法提供具有实际价值的内容，那其实一无是处。

尽管Windows给桌面输送了内容，微软却并没有创建内容。内容是由其他方生成的：联合新闻服务商、电影制片厂、独立制片人以及数以百万使用文字处理程序、电子表格和图形编辑器的用户。Windows的作用只是给这些内容提供传输的载体。Windows这些华而不实的图形用户界面和桌面提供的只是一些闹哄哄的玩意儿，并没有什么实质性的东西，这所谓的"实质性的东西"应该就是应用程序的内容。当然，如何传输内容是很重要的。只是，相比你要传递的内容，图形用户界面的重要性相形见绌。

是的，内容为王，这是事实。因为即使电脑不存在，供应商仍然可以找到其他办法让内容能够传达到用户手中。一开始，内容在村与村之间口耳相传。随着印刷机的出现，内容以报刊、书籍和文字的形式从一个城市传播到另外一个城市。后来，广播电台和电视台为我们输送内容，成了为大众群体服务的大型媒体。万维网和因特网也只是下一种输送所有重要内容的手段。

就算微软、Linux和网络都不存在，内容仍然会自己跑到你面前，它就是这么的重要。如果真的是很重要的信息，你无论通过什么途径都会收得到。总会有人因为商业或意识形态的原因去传播信息。他们会千方百计地让你能够获取内容。

因此，有没有图形用户界面并不重要，真正重要的是那些被传达的信息。如果你对看到的内容并不感兴趣，就会对它们置之不理并寻找下一条信息。

不过，某些类型的传输机制难道不比其他的机制能够更好地提供内容吗？是有这种情况。你也许觉得任何形式的资料都比不上视频，但事实可能会让你惊讶，即使在当今这个时代，视频的力量与文字相比依然相形见绌。

让我们来看看三种主要信息传送的机制：视觉、听觉和文本。通过对这些形式的仔细考量，我希望你会明白，为什么在CNN和MTV占据主导地位的年代，Unix

这种主要面向文本的系统仍然无比强大。

我们可以先研究一下视觉媒体，它是三种媒介中表现最弱的。咦？什么意思，三个中最弱的？大家都知道视频是一种强大的媒介。再坚持一分钟吧，跟着我的思路，翻到下一页你可能就会有不同的想法了。请记住，我们是在将Windows这种视觉型的操作系统与面向文本的操作系统Unix作过比较之后才得出的结论。

10.1.1　视觉内容："用自己的眼睛去看。"

毫无疑问，视觉媒体的确非常强大，尤其擅长激发观众的情感。当你在观看某些东西的时候，就好像你和屏幕之间建立起了一些联系。无论是积极的方式还是消极的方式，大部分的视觉效果都会引发你的情感共鸣。

有两种视觉媒体，静态的和动态的。静态视觉媒体包括图画和照片。在万维网和互联网时代，静态图像比比皆是。人们访问的几乎每一个网站都包含有静态图像。很多大型商业网站会显示所谓的"横幅广告"，这些商业化图片的目的是引诱你点击并将你带到另一个网络站点，也许你会在那里花点儿钱购买一个纯粹因人口统计学特征而为你推荐的产品或是服务。

制作静态图像并不难，这使得它成为一种相对便宜的媒介。如今，我们可以使用数码相机迅速拍摄照片，或是从视频源文件中获取那些静态的图像帧。其他图形文件则可以由Adobe的Photoshop或是一种流行的The Gimp的开源应用程序产生。

静态图像的一个缺点是，如果没有关于其内容的一些文字说明，它们可能很难进行归类和检索。举例来说，假如要求你去寻找一张伐木工人砍树的照片，你可能要花上很长时间在数千张照片中查找才能找到。大批量照片处理服务通过提供关于照片内容的文字描述简化了这个任务。这些服务的消费者随后便可以在高级搜索引擎的辅助下，通过搜索文字描述来查找感兴趣的照片。

在典型的计算机桌面屏幕上，静态图像通常作为图标来表示一个要执行的功能或是被操纵的对象。因为人们识别图像的速度往往要比识别文本的速度快，在使用图形用户界面的时候，用户经常会发现图标加快了他们的操作速度。问题是，除非知道这些图标的含义，否则实际上图标反而会影响你的操作速度。你甚至可能需要去参考图标附带的描述文本。

今天，动态视觉媒体主要是指视频，更具体地说就是电影。虽然随着电影和电视的出现，这两种媒体已经非常成功，可它们并没有像人们料想的那样获得和文字一样的影响力。比如说，虽然很多人喜欢看电视晚间新闻，可令人惊讶的是，还是有很多人更喜欢阅读报纸或杂志上的文章。尽管很长时间以来，电视已经成为许多家庭里最受欢迎的一种娱乐方式，然而随着以因特网为基础的新型娱乐形式出现，它的地位却开始被更具交互性的追求所撼动。过去，视频可能终结了广播明星，但现在互动点播却使得电视明星成为明日黄花。

制作静态图像的成本相对便宜，可制作哪怕是低质量的视频内容所花费的代价却都相当可观。就算只是制作视频短片也往往需要大量计算能力和存储空间。将视频演示集成在一起就更需要具备相当的技能水平，这可比使用相机要难多了。此外，按照今天的视频格式，一部两小时的电影所需的存储空间超过了500MB。但相比之下，数百甚至数千张静态图像需要的存储空间也超不过1GB。

当需要归类和检索的时候，视频也有着与静态图像相同的缺点。我们仍然需要某种文字说明来检索特定主题的视频剪辑片段。可实际视频剪辑片段的存储需要大量空间，这进一步加剧了问题的严重性。试想一下如果某个人有10万个不带标签的视频剪辑，想要定位某个内容为“聪明的动物要比蠢笨的人智商更高”的视频就会是一项艰巨的任务。

综上，视觉媒体虽然很明显是一种强大的、有活力的内容，但它们仍然需要文字描述的“辅佐”。如果没有那些能够用于搜索和检索的文本，制作图像和视频将是一个费力的过程，而且制作成本也会飞涨。因此，在没有文字描述的情况下，我们只能尽量减少影片的数量。虽然用来标记视觉媒体的文字并没有很“好”的视觉效果，但它却物美价廉。这也正体现了前一章中阐述过的“更坏就是更好”的准则。嗯，也许这其中有一个固定模式？你可以自己在脑海中想象一下这幅画面，下面让我们去研究一下有声内容。

10.1.2 有声内容：“听得到吗？”

有声内容是另外一种多媒体组件，它的历史比视频还要悠久，但是还是比不上静态的视觉媒体。它包括*原始音频*、*音乐*和*语音*。接下来简单地看一看其中的每一种文件。

*原始音频*指的是那些人类的耳朵能听到的声音。其中的许多声音早已被录制在

磁带上，或是被转成数字化形式然后存储为计算机上的音频文件。它们通常被称为“声音片段”，这些声音的范围很广，从洛杉矶东部枪战的疯狂喧嚣到高山湖泊日出时人们几乎感觉不到的天籁。

对于我们都熟悉的音乐来说，用任何的语言来形容都显得多余。音乐是一种没有国界也不在乎年代的沟通媒介，它就如同你踏上人行道那样自然。此外，关于音乐创作和欣赏的书可谓是汗牛充栋。

音乐具有很多与视觉内容相同的特征。它的制作成本可以非常低廉：比如现场表演；也可以非常昂贵：比如在那些耗费巨资购买设备的音乐工作室中精心录制。对于视觉和听觉内容来说，它们的质量往往取决于人们到底愿意花多少钱。

音乐和视觉内容一样都能够将情感传递到个人。正如人们在看各种电影时体会到的喜悦、恐惧或是悲伤的感觉，通过各自喜爱的歌曲，人们同样也能获得一份相通的喜怒哀乐。这些年来，很多电影都会包含一些由著名音乐人录制的原声音乐。随着时间的流逝，两者之间的界限正在变得越来越模糊。

最后，语音中的声音内容可以用文本形式来表示。人们说的话语包括各种形式，从新闻报道、电台访谈节目到历史性的演讲。通常都是这样的状况——听众的时间被分割成两个部分，一方面他们在接收所听到的内容，另一方面他们还在完成其他的一些任务，比如坐车或是做家务。而且，在印刷媒体中的信息也会转换成口语内容，比如说，《今日美国》（*USA Today*）报上的文章会在地方和全国的新闻广播电台中播出。

语音的一个缺点就是它对语言非常敏感。发言者使用的语言必须与听众说的语言一样，在一些情况下甚至得是同样的方言。否则大家就会沟通不畅，语音的价值也会大打折扣。更糟的是，在某些情况下语音甚至会存在文化偏见，因为根据不同的上下文、发言者和听众的背景，某些词语会代表不同的含义。实时翻译服务确实存在，但聘请优秀的口译人员可是价格不菲，而且语音的即时性也导致人们不可能将所有的发言内容全部翻译成自己的母语。

想让所有这三种有声内容具有价值，我们需要对它们进行分类和编目。所有的音乐制作、声音网站和录制的演讲都需要有一些文字性的记录，以备后续快速检索。例如，假设一个人收集的音乐作品集包括一千多张没有标签的光盘。你可以想象一下在某一时刻，这个人如果想找一张特定的光盘来听听会有多沮丧。即使是那些磨

损或难以辨认的文本标签也会有助于我们对每张光盘上的内容进行分类。

即便音乐存储在其他的媒介里，比如个人电脑中，按文字来归类仍然是必须要做的事情。假设这些音乐并不是存储在光盘中，而是存储在计算机上的具有类似文件名的10 000个歌曲文件，那显然不方便检索。相比之下，如果所有的歌名都存储为纯ASCII文件的列表或简单的数据库，那查找喜爱的歌曲就会是小菜一碟。无论文字资料目录是印在标签上或是存储在数据库中，我们都不能低估它们的价值。

从这一点我们可以明显地感觉到，视觉和听觉媒体善于传递图像信息和表达情感，但它们却无法单独存在。一旦其内容超过一定的数量，比如说几十个，就需要文本数据目录的辅助。事实上，如果没有能够用于识别这些媒体的文字性陈述，你能掌控的电影、演讲和音乐剧的数量将会非常有限。文本是音频和视频媒体的基本推动力。这也就是为什么在所有内容中，文本才是那个最强大的形式。

10.1.3 文字内容："视频可以终结广播明星，却消灭不了小报。"

虽然视觉和听觉的内容能够带动我们的情感，可最终还是文字内容吸引我们去了解更多的信息。一个网站可能会有眩目的图形效果和最酷的声音，但这些都无法永远维持。除非这个网站有你感兴趣的书面内容，否则你不会流连太久，很可能以后都不会再访问这个网站。

许多人曾经认为电视的出现会危及广播的生存。在大多数情况下，广播电台只能屈从于电视的冲击。大家早已不会再匆忙赶回家去收听最新的无线电广播。人们也许会在开车的时候听听收音机里的新闻，但一回到家他们就会打开电视看CNN频道或是观看他们喜爱的电视节目。电台的吸引力已经不复从前。在人们可以通过视频接收一切的时候，为什么还需要通过电台去获取那并不完整的信息呢？

虽然视频几乎将广播明星终结，可它还是发现自己的垄断地位被一种更强大的力量夺走，那就是——互联网。过去一度爱看电视的人们现在却开始沉迷于万维网上的互动体验，也许这种沉迷的程度比以前看电视还要厉害。那么，什么是网上最流行的内容形式呢？我可以给个提示：它不是音乐文件（尽管这些确实很流行），也不是视频剪辑片段，而是文字，纯粹而简单的文字。

人们之所以涌向互联网和万维网是为了获得文本内容。网络是人类有史以来最大的图书馆，而且一直都在变化。无论你的兴趣是什么，不管它是多么的模糊，你

都能在网上的某个地方找到关于它的内容。而且，几乎所有内容都是文字。有一个视频片段或音乐文件，就会有数千个包含最基本图形和大量文字的网页。

文本至上。为什么？尽管受到广播、电影和电视的冲击，报纸依然存在并还在蓬勃发展。虽然每个人的第一念头都想到了计算机可能会把我们的世界变成一个无纸化社会，不过今天人们每天收到的垃圾邮件数量恰恰证明了这样一个事实——文本很重要。不管是打印文本还是在线文本都仍然生机勃勃，在你的邮箱里，文本邮件的数量要多于诸如Caltiki the Immortal Monster[①]之类的视频文件。让我们来研究一下为何文本内容是如此的强大，为什么这些年来它会一直持续蓬勃发展。

*文本是一种用来存储和传播思想的廉价方式。*就存储空间而言，文本内容比起音频或视频小多了。例如，我这本书占用的空间还不到1 MB。用CD或DVD存储它的代价远小于一美分。如果我创作一本有声书，它需要的存储量将会数百倍于文本文件所需要的空间。如果将它制作成视频演示，花费的代价将更加昂贵。

通过网络传送文字内容的花费也不高。就算网络带宽已经大大地增加了，但因为文本文件的大小比起音频或视频文件还是具有绝对的优势，所以文本文件的传输永远都是那么的便宜。在快速网络里，整本书的传输只需要几分之一秒。而其他内容形式的传输要更久一些，有时候这个时间可能还会更长，这取决于文件的大小和网络吞吐量。

在2002年美国盐湖城冬季奥运会上，Qwest Communications公司构建了一个传输能力超过388 T比特/秒的网络[②]。用直观一点的说法来形容，它差不多每秒能够传输2倍于整个美国国会图书馆馆藏图书的内容，这个图书馆目前存储的书籍超过了17万册。有了这样的网络传输能力，理论上我们可以在几分钟内（或在更短的时间内）就将世界上的所有书籍传输完毕。音频和视频的传输都无法达到这样的高效率。使用文本内容，你就再也不用担心是否拥有足够的带宽了。

*人们可以高效地对文本进行索引和搜索。*即使每秒钟能够传输2倍于国会图书馆馆藏图书的容量，如果你不知道自己具体要查找什么书，那又有什么意义呢？这里就要谈到今天那种速度超快，能够对庞大的文本量进行查找的搜索引擎了。在写作本书的同时，最流行的互联网搜索引擎Google索引的内容超过了20亿个网页、7

① 你可以到在线电影资料库 http://www.imdb.com 去查看更多关于这部恐怖电影的资料。

② 参见2002年1月14日出版的期刊*The Telephony*。

亿多的新闻组、公告栏信息、3 500万多非HTML（即非网页）文件。尽管数量如此之大，可那些稍微有一些前瞻性的人几秒钟内还是能在网络里完成大海捞针的任务。你可以尝试着与你的家人去搜搜那种高达20亿页的相册。查看图像也许很有趣，但如果没有文字说明的话，查找它们的速度可是相当的慢。

我们有必要对Google的技术架构一探究竟。有趣的是，Google并没有采用主流Unix系统供应商的大型RISC服务器。它使用的不过是10 000多台杂牌PC机，它们都是在Intel架构基础上运行着Linux操作系统。这些PC机是机架式的，并通过100 Mbps的网络连接在一起。因此，这个世界上最快的全文搜索引擎其实只是运行在廉价的PC机集群上，运行的也是免费开源操作系统。

小不仅是美，还更便宜。读者可以自己算一算，如果10 000台计算机使用的都是微软的Windows操作系统，那许可证的费用要多少钱。

Google和其他主要的文本搜索引擎还有一个有趣的功能，那就是将外文网页即时翻译成许多其他语言的服务。Linux机器上的KDE Konqueror网页浏览器就有一个“翻译此页”的菜单选项，它会跳转到AltaVista的网站，并将你当前的网页转换成任意流行的语言。但是别急，我们的步子其实用不着迈得这么快。

通过机器将文本翻译成其他语言的花费并不高。自17世纪以来，机器翻译文本就一直是语言学家的神圣目标。将外国语言文字自动翻译成自己的母语多年来一直都是无数科幻小说的主题。不过，这个想法直到20世纪后期才成为可能，而且变得切实可行。今天，互联网上功能强大的服务器将机器翻译的文字变成了商品。几乎所有的主流搜索引擎都提供最流行的西方语言甚至一些亚洲语言之间的转换服务。在许多情况下，科幻小说中描写的情景早已成为了科学事实。

机器翻译（Machine Translation）的表现还远远不够完善。但相对于聘请一位全天候的真人翻译，机器翻译的成本相对低廉并且颇为有效。它无法提供真人翻译那样的准确性，特别是在处理大多数习惯性用语的时候，它的表现尤其地差，除非这些习惯用语有着明确的表述。因此从某种意义而言，机器翻译的表现不如真人翻译。但是，我之前不是谈到过廉价但有效果的事物吗？是的，更坏就是更好。

那电影呢？请问在这种情况下，文字还会比声音和视觉内容更具有优势吗？难道不是通过观察人物的动作和表情，人们更能了解外语电影讲述的故事吗？是的，这一点毋庸置疑。再次强调一下，视频和音频往往更能表达情感。但是，比如说想

真正地理解电影《卧虎藏龙》蕴含的微妙意义，你还是得去看一看字幕。文本至上，它是理解的关键。

文本可以非常准确地传达意思。当人们第一眼看到一个新的图形用户界面时，会敏锐地意识到采用图标和图像来识别事物是非常不准确的。如果你在过去看到过类似的图形用户界面，那的确能够传承以前所积累的知识，了解屏幕上的那些图标和图像都代表什么意思。关于文字处理器相关文件和程序等的识别，你不会有太大的困难。在帮助快速识别你所拥有的资料时，图标和图像非常有用。

不幸的是，图标本身只能提供一些象征性的意思。当你有不止一份资料并且需要定位到特定实例的时候，就会有问题。在这个方面，你需要更加明确的标识符，而不是那些看起来千篇一律的图标。你可能记得住，点击左边的文件图标是打开你的博士论文，点击右边的文件图标则会开启好友发给你的笑话集。但是，如果文件夹里包括一打类似的图标，你还记得住它们都是干什么用的吗？如果你常用的互联网搜索引擎返回关于“Linux内核”的搜索结果，每一页都包括二百个文件图标的话，这会有什么用处？在你看完这大概二百万个返回结果之前，你很可能会更愿意付50美元来脱离这个“图标地狱”。

在可直接操作的用户界面（如微软的Windows）里，假如你拥有的数据工作集规模非常小，图标和图像就会非常有用。可要是你有十几个或更多文件的话，图标下面就需要文字注解以帮助你识别这些文件的内容。最终，这些文字注解其实比图标更重要。有经验的Windows用户通常会设置好浏览器或文件管理器，让这一类型的应用程序能够显示文件名列表，而不是大图标。

Windows以文件夹视图显示的文件树尤为繁琐。很不幸，这一点颇具讽刺意味，因为抽象的文件树在Unix和Windows的世界中其实表现得可圈可点。但在Windows的环境下，随着你逐步深入到多层文件夹，通过点击图标来浏览文件树就会变得愈发地耗时耗力。Linux和Unix上的高级用户发现，严谨的文字表述是一种锦上添花的做法，因为人们可以将文件树的文字表述轻而易举地嵌入到像grep这样的搜索命令里，比如下面这条命令：

```
find . | grep myfile
```

在这种情况下，find命令会返回当前文件夹所包含文件夹和文件的列表。find命令是递归操作的，因此它会打开目录中的所有文件夹以及文件夹内的子文件夹，

如此循环查找。grep命令会在所有的文件名或是名字列表中去选择那些名字包含myfile字符串的文件。

上述命令中这种详细且深入的文件路径有一个优势，就好像文件名：/home/mike/eventcalendar/2004/May/19/Bahamas/Freeport/Princess/Towers.txt。你可以在前面那条命令里用Towers.txt来替换myfile，很快你就能找到这个文件。当然，Unix系统允许你将一个命令或一组命令的输出结果设置为另一个命令的输入。以下是在Unix环境中快速定位和编辑Towers.txt文件的办法：

```
vi `find . | grep Towers.txt`
```

想要在Windows环境中执行以上命令完成的动作，你不得不从/home文件夹开始，遍历每一个文件夹直到你最后发现Towers.txt这个文件。然后，你还需要去调用该文件的文字处理程序。现在想象一下，如果每个文件夹中包含的文件不是十几个，而是数百个。那么，这样一级级地浏览每一个文件夹直到找到通向所需文件的正确路径[①]，将会是多么繁琐的任务。

如果觉得这还不够快，你还可以考虑使用Linux中的locate命令。它的使用模式与find命令几乎相同，不过它用到了一个参照指针索引的数据库，几乎可以在瞬间就定位到文件和文件夹。人们只需花上几分之一秒就能够在成千上万份文件堆中找到需要的那份文件。

用文本来替代图标和图形用户界面，你就可以轻松地处理任意宽度和深度的文件树。这并不是说在Windows下无法操作大的文件树，只是更麻烦而已。随着数据集规模的增大，点击越来越多的对象就成为艰巨的任务。另一方面，使用文字能非常精确地说明你要寻找的文件，就算它被嵌套在深达多层的文件夹里也没大碍。

不熟悉这个概念的人们在看到高水平Linux用户对着屏幕上的终端窗口键入文字时，经常会难以置信地摇头。他们认为终端和电传打字机就像是远古时代的恐龙遗迹。他们很困惑，为什么在我们这个时代这些“尼安德特人”[②](Neanderthal)还不使用鼠标，像在Windows环境中那样通过点击图标来进行操作呢。

① 是的，你可以在Windows资源管理器里面使用“查找文件或文件夹”的选项来查找Towers.txt。在它提示你往空白的对话框里输入文字的时候，Linux上的用户早已经调用vi来编辑文本了。正是这些多余的键盘和鼠标的操作步骤放慢了你的节奏。

②尼安德特人曾是欧洲大陆上盛极一时的古人类，有着20多万年的辉煌历史。——译者注

答案很简单。文字的反应速度更快。迟早，Linux用户都会在终端窗口中使用文本，因为这种做法更有力量。人们点击图标和图像的动作再快也不过如此。除此之外，文字还有一个被人们忽略掉的优势——严谨。这并不是说Linux用户永远不会去点击在KDE或Gnome图形用户环境的图标。只是，当你需要快速并高效的操作时，图形用户界面的反应速度还不够快。

Windows用户成为高级用户后，他们必须加快点击的速度。另一方面，Linux的超级用户只需要键入少量的内容就能得到更好的效果。嗯，你是想点击得更快还是想输入更少的内容？这由你自己决定。

由于数据抽象这个概念，因此文字在处理大量内容时更为高效。小图标通常是用来代表一个较大图像的微型缩影。瞄一眼小图标，你大概就会知道整个图像的全貌。文字并不具备这样直接的表现形式。然而，它有更好的作用。

例如，假设使用的文件路径是下面这行文本：

```
/home/mike/files/documents/financial/tax_returns/2002/return.tax
```

瞄一眼你就能反应过来，该文件的路径意味着这是我2002年收入所得税的申报表。此外，这个路径还表明了：(1) 这可能是我存储在这个系统上的诸多税务报表之一；(2) 我可能存储了多种财务档案；(3) 一般我可能会保留一种以上的档案；(4) 除了档案之外，我还可能保存了一些其他类型的档案；(5) 有可能其他用户的文件也保存在这个系统上；(6) 系统上的/home目录可能是保留其他大多数用户文件的区域。

单从文件路径中你就可以收集到如此多的信息。想用图形来表示同样的效果（并且完全没有文字作为辅助）其实很难，你难以通过不同的图标来显示文件路径中的每个组件，用户难以记住每个图标代表的意思。人们可能最终还是要给每个图标添加辅助性的描述文字，以帮助识别每个图标具体代表什么内容。当然，在这一点上用户还是要回到老路，也就是使用文本来描述内容。

我们可以永无休止地继续举出一些用文字表述数据抽象的例子，但我想你应该已经领会了这个要点。很遗憾，使用图标或图像的图形表示法无法胜任将数据分类和抽象的任务。

正如我们在前几页中看到的，音频和视频内容虽然在情感上具有极大的感染

力，但它们在我们的计算机系统里起到的只是辅助作用。就几乎所有的内容处理任务而言，文本是首屈一指的载体。计算机系统里使用最多的就是文字内容。如果没有文字内容，大多数计算机将成为我们生活中很少使用的华而不实的玩具。

但是，将Unix及其变种与微软的Windows进行对比为什么会如此重要呢？这是因为Windows主要是一种视觉化的系统。微软已经反反复复阐明过这一点，图形界面的人机交互是Windows哲学的核心要素之一。

作为文字处理系统的能力，则是Unix及其变种的力量源泉。文字是一种最深刻的传递内容的形式，典型的Linux或Unix发行版上有数百个专门针对获取、创建和操纵文本而设计的程序。目前，世界上没有任何其他系统能够在文本处理能力方面与Unix抗衡。

在这一章中，我们已经了解了微软Windows操作系统的哲学，它是一个以图形作为主导的系统。我们已经看到，虽然在最开始的时候，图形用户界面能够帮助人们克服对电脑的恐惧感，可随着数据量的增加却会导致一些实际问题，同时也不利于让新手用户成长为高级用户。在文本内容成为用户的首选媒介时，视觉化的操作系统实际上妨碍了人们对内容永无止境的追求。

这并不是说，图形用户界面天生就是邪恶的。它们自有其用途，而且有时候使用起来也很方便。Linux/Unix与Windows之间的主要区别在于它们的实现机制。在Windows系统中，用户使用图形用户界面与系统进行交互。在Linux和Unix中，图形用户界面只是与系统交互的方式之一。区别就在于此。

前面的阐述已经为我们一般性的理解打好了基础，现在是时候重新审视一下Unix哲学的具体准则及它与Windows的关系了。我们将看看Windows是符合这些准则还是与之背离。这部分内容只是点到为止。现在，你可能已经了解Windows的组织架构，接下来我们只是将这些要点再复习一遍。

准则1：小即是美

早在Windows 3.1时期，微软的Windows只是一个规模颇小的图形用户界面，它建立在规模更小的MS-DOS之上。随着时间的推移，Windows逐渐成长为一个庞然大物。尽管微软使用了有助于减少空间消耗量的共享库，可Office和IE浏览器这样的应用程序还是因其庞大规模而赢得了一个流行的绰号——臃肿软件（bloatware）。

如果你认为它们规模庞大是因为提供了相当多功能的话，我想请你再翻阅一下之前关于人类的三个系统的讨论。这些应用都可以算是受“第二个系统”综合症所累的典型例子。当微软终于认识到这些应用程序需要变得更小而不是更大的时候，它就会开始走上通往“第三个系统”的康庄大道，这才是三个系统中最好的。与此同时，我们还需要反复审视应用程序的数百个功能，从而提高执行这些应用程序核心任务的能力。

准则2：让每一个程序只做好一件事

这条准则谈到了计算技术中的小工具或组件方法（component approach）。虽然微软也有如OLE和ActiveX等的组件方法，可在一个重要方面上它却与Unix哲学截然不同。Unix组件可由用户操纵并集成，而微软却极力想掩盖这些组件存在的事实。Unix的做法是告知组件的用户并鼓励他们充分利用这些组件，但Windows组件的做法却恰恰投射出它的一个错误设计理念，也就是假设用户害怕电脑，而且认为他们不应该接触到计算机的内部机制。

准则3：尽快建立原型

微软在这方面表现得相当不错，因为它的工具（比如Visual Studio和其他IDE）都推行这种“快速建立原型”的做法。我曾经参加过一个由微软销售工程师做的展示会，他向我们介绍了如何能在短短五分钟之内使用微软的工具创建一个基于Web的数据库应用程序。花五分钟建立好一个原型可以节省很多时间，让开发人员能够与用户讨论该如何最大程度地改善应用程序。请注意，VMS习惯在编写任何一行代码之前都要经过彻底的讨论，在这方面微软的做法已经与VMS系统背道而驰。看，微软做得好的地方，我是不会吝惜我的表扬。

准则4：舍高效率而取可移植性

为什么微软和英特尔会被统称为Wintel垄断集团呢，这其中有一个原因。多年来，微软各种Windows操作系统都执着地与英特尔x86 CPU产品线捆绑在一起。在孜孜不倦地追求让图形用户界面能够最高效运行的同时，微软一直与英特尔的硬件紧密结合。随着Windows NT的推出，微软貌似终于看到了可移植性的价值所在。Windows NT能够在四个架构上运行：英特尔的x86、DEC 的Alpha、MIPS的R4400和摩托罗拉的PowerPC。然而，Windows XP却来了一个大倒退，它只能运行在英特尔的x86架构之上，反观Linux却能够在当今所有的架构上运行。微软是否已经真正

领会到了可移植性要优于高效率的真谛？从Windows XP的种种迹象来看，恐怕是没有。

准则5：使用纯文本文件来存储数据

微软在这方面的表现乏善可陈。虽然有一些Windows系统配置文件采用的是ASCII文本，但微软的世界中仍然喜欢大量使用那些极具隐蔽性的二进制文件。比如说，Windows将关键的配置信息保存在注册表中，那是一种极易损坏的二进制格式文件。开源行动的倡导者如自由软件基金会（Free Software Foundation）的Richard Stallman早就义正言辞地抨击过微软的Word、Excel和其他关键应用程序仍在使用二进制文件格式这一顽固行径。在计算机技术的所有领域，人们普遍会采用XML（可扩展标记语言）这样基于文本的文件格式。撰写本文时，微软计划将Office的二进制文件格式改为XML。而反观另一方面，基于Linux的Office工具使用XML都好多年了。

准则6：充分利用软件的杠杆效应

一个坚持闭源、采用专有开发模式的公司如何能够利用软件的杠杆效应？嗯，微软其实有一个办法，它可以吸收各种开源开发项目中的好想法，并将它们纳入自己的产品。为了自己的利益，微软其实早就这么做了。但只有保持一条双向流通的软件开发道路，才能充分发挥软件的杠杆作用。微软必须让其他人能够轻而易举地利用它的源代码。这意味着将Windows源代码公开以供其他人的使用和改善。这么做将会产生一系列新的Windows应用程序，其中大部分都能同时被Windows和Linux所用。我们已经讨论了这种情况发生的可能性。

准则7：使用shell脚本来提高软件的杠杆效应和可移植性

为了使用shell脚本，就必须有一个shell命令解释器。微软拥有一些类似的应用，比如Windows Scripting Host和MS-DOS批处理机制。而且由于这个世界上的Perl粉丝（或者是狂热分子）实在是太多了，因此现在微软的平台上也有了可用的Perl版本。问题是，Windows Scripting Host和MS-DOS批处理文件的命令集非常有限，而Perl却拥有一组丰富的模块，前提是你愿意花时间研究并安装它们。此外Windows在脚本环境方面的表现很平淡无奇。因此，默认状态下，用户通常都是通过图形用户界面与系统进行交互。这就意味着Windows能够执行的任务只局限于那些有先见之明的程序员编写出的图形用户界面应用。在Windows上，人们根本就不会有那种

灵光一现的举动，临时起意去编写一个脚本。

这影响到Windows程序是否能够以最佳速度运行。图形用户界面应用程序需要时不时地去点击按钮并从列表框中作出选择，这就迫使程序必须与真人进行交互，而不是与其他程序。正如我们前面讨论过的，这种人机互动成为制约系统达到最佳性能的因素。除非程序是故意设计成无需人工干预，否则它的运行速度就受限于人按动鼠标按键的反应速度。大多数Unix程序却可以由脚本来驱动，这就意味着它的速度是由机器本身的运算速度决定的。

也许有人会争辩，不需要人工干预而运行的操作系统只不过是一台服务器。从这个观点出发，一些专家试图贬低Linux作为主流操作系统的潜力，而将它的角色归类成无大作为的服务器。他们认为Windows卓越的图形用户界面将继续统治桌面领域。其实，这些批评家的论断根本就没有抓住要点。Linux是一款非常优秀的服务器，但它也可以在所有的计算领域里都表现良好，包括桌面。正确构建简单的桌面环境只是服务器的另一个任务。人们只需要数数Linux上桌面环境的数量，就会了解Linux早已成为了一款高水准的桌面服务器。

准则8：避免强制性的用户界面

从本质上来说，图形用户界面就是CUI。它几乎完完全全地吸引着用户的注意力，除非满足图形用户界面应用程序的输入要求，无论是从选项列表中选择还是按下“确定”按钮，否则用户甚至无法去做别的事情。为了解决这个问题，Windows系统允许用户打开多个窗口，从而使得用户能够同时与多个应用程序交互。因此，从这个角度而言，这些窗口系统至少可以假装自己遵守了“避免使用CUI”这一准则。

Windows程序员在编写大部分（不过，幸好不是所有的）Windows应用程序的时候，就假设用户只想与单个应用程序进行交互。他们忽略了一个事实，Windows不单是图形用户环境，同时也是一个窗口系统，很可能会有多个应用程序来争夺用户的注意力。有一个最明显的例子，也就是那种迫使用户与之交互的弹出对话框，人们必须点选里面的某个选项。我们称这一类对话框为模态对话框。许多现代Windows应用程序，尤其是那些从新千年才开始编写的程序对此状况已经颇为敏感，它们会尽量避免陷入这种模态对话框的困境，尤其是在用户需要与多个应用交互来完成某个给定任务的时候。然而，这种老旧的互动模式有时候还依然存在。

有一种方法可以避免CUI，那就是将用户和应用之间的互动视为客户端和服务器端之间的通信。KDE DCOP（Desktop Communications Protocol，桌面通讯协议）就体现了这种思想。它是由KDE开发人员Matthias Ettrich 和Preston Brown共同开发的，构成了一种通用进程间通信设备的基础[①]。与其他处理非人机交互的进程间通信协议（如RMI）相比，DCOP的不同之处在于它的主要目的是为了在KDE桌面环境的基础上处理人机交互。有了DCOP，程序就可以通过标准API轻松地与其他程序交流，其中很多程序都提供图形用户界面。然而，因为任何理解DCOP协议的程序都可以在KDE用户环境中运行，这使得运用脚本工具来驱动桌面对象和应用成为可能。在使用DCOP的情况下，KDE客户端实际上并不需要知道它传递的是什么消息，只要它能使用DCOP协议即可。

如果你认为这种做法听起来和“所有程序都是过滤器”的概念非常接近，那你就理解了这一思想的精髓。不管通信方式是Unix管道还是DCOP连接，本质都是相同的，小程序和小程序又可以交互了。

在这里，我想继续强调一下，Windows其实不太容易适应这一类脚本工具。但我有一种感觉，我的“说教”只有那些日益增大的KDE开发人员群体才听得进去，他们早就发现了一个建立桌面环境的更好方法。不过，它算是一个不错的引子，将我们带入下一条准则。

准则9：让每一个程序都成为过滤器

基本上，在Windows的世界里，这条准则是一个非常陌生的概念。Windows的做法往往是这样的：“让我们一起努力构建一个规模庞大且有着无数功能的方案，如果我们的销售策略够高明，这个产品就会卖得很好。”一个公司的确能够通过销售这类产品而成为软件巨头，但这并不意味他们的做法就是对的。比如，微软Outlook无法与其他程序进行良好的交互，这是因为开发人员并没有将程序看作是过滤器。我们可以对比一下之前讨论过的邮件处理程序MH。MH的功能集也很全面，用户和开发人员可以轻松地为MH套件添加新的组件，而Outlook却是一个规模庞大且极不灵活的环境。Outlook开发人员想出了多少功能并不重要。他们始终是局限在自己的一方天地里，因为他们只是Outlook程序的开发人员。其实，Outlook之外的世界也有很多奇思妙想。

① 参见http://developer.kde.org，KDE 3.0的架构。

如果将程序视为过滤器，你就会选择编写在未来有改进空间的开放式程序，这样的程序不仅对你和你的公司开放，其他任何有着不同看法的人也可以改善你的软件。这样的话，你就能够受益于他人的努力，正如他们也能受益于你的努力一样。

关于微软Windows这一章就讲到这里。但在我们准备进入下一章去讨论开放系统之前，我还有一些想法，可以作为本章的总结。

我们已经研究过计算机行业中的“羊群效应“。在撰写本文时，微软显然成为了大众的热点。正如我们曾经有过一个所谓的“IBM机器就是拿铁饭碗”的时期一样，今天我们也可以加入大力鼓吹微软解决方案的阵营，从而获得职业保障（无论如何这会持续一段时间）。尽管微软实际上还是有一些相当不错的软件（我并不讳言这一点），不过这些软件却是由微软和有限几个曾被微软视作是盟友的公司一起推出的。这个世界上还是有其他开发人员、其他公司和其他声音，他们应该拥有展示自己创意的机会。

尽管微软杰出的市场营销能力使得它对世界产生了巨大的影响，但它并不是计算机世界中独一无二的力量。就算一大群人坚持认为微软的Windows是计算技术中最正确的手段，我也可以找另外一堆人来反驳这个说法。

我找来的这一堆人看起来就像是一群企鹅，他们戴着红色的帽子和黄色的星星，他们像变色龙一样变化多端，他们说一种奇怪的语言，里面包含着诸如grep、awk和sed这样的字眼。

他们信奉一种叫做Unix的哲学。

第11章 大教堂？多怪异

11

“起初他们忽视你，尔后嘲笑你，接着打压你，最后就是你的胜利之日。”

——甘地[1]

2002年10月1日，当滚石乐队（Rolling Stones）发布专辑*Forty Licks*的时候，每个人都认为这张唱片应该能直接问鼎下一周唱片排行榜的冠军，但它却没能成功，只是屈居第二。滚石的这个故事证实了一个道理，你无法总是得到你想要的东西。这张集合历史上最伟大摇滚乐队热门曲目而成的精选集之所以未能问鼎，是因为广大听众情不自禁地爱上了另一张精选集，它集结的正是伟大歌手猫王Elvis Aaron Presley的热门曲目。这就是2002年9月24日发行的名为*ELV1S 30 #1 Hits*[2]的专辑。

心存疑问的人们可能想知道为什么像猫王这样的老家伙会让滚石乐队无法得偿所愿呢。为什么他们就无法摆脱猫王的阴影？为什么是滚石败给了猫王，而不是猫王败给滚石？

答案很简单。滚石乐队的音乐是在“大教堂”里完成的，而猫王的歌曲却是借用了美国音乐“集市”中的种种元素。咦？在你觉得这个说法是如此怪异之前，让我来解释一下吧。也许你会明白为什么在教堂里哭泣的是滚石乐队而不是猫王。

从滚石乐手Mick Jagger和Keith Richards处于音乐生涯的早期巅峰开始，他们就在一起度过了无数个不眠之夜，创作出一首接一首让大家热血沸腾的热门单曲。他们的作品只有摇滚这一种风格，但我们还是很喜欢。Jagger和Richards似乎做不了别

① 莫罕达斯·卡拉姆昌德，甘地（1869—1948），是印度民族解放运动的领导人和印度国家大会党领袖。——编者注

② 是的，这个名字“ELV1S”绝对不是笔误。CD名字上的第四个字母确实是数字“1”。

的。他们只做了一件事情，可他们完成得很好。

然而，对于他们所具备的创作天赋来说，他们的作品只是一种“闭源”的专有做法。我们几乎很少听得到滚石与任何第三方一起创作出来的歌曲。几十匹野马也无法将他们从心中的大教堂里拉出来，去和别人一起开展任何开源合作。他们可谓是自身NIH综合症的受害者。

然而，猫王的妙处就在于他这种大杂烩的集市化创作方式。他喜欢所有令人热血沸腾的音乐元素，他的歌曲风格包罗万象，从摇滚、蓝调、乡村、福音[①]、民谣到流行曲调，一切都取决于他当时热爱的是哪一种。他的魅力源自他多变的风格。不管是在大屏幕上还是工作室的麦克风前，或是去参加著名的电视节目Ed Sullivan Show，猫王都是激情四射，愿意表演任何节目。

猫王也是深谙重用之道的大师。在Mick Jagger和Keith Richards忙于创作原创音乐的同时，猫王却正忙着借鉴别人的音乐，并将它们打造成自己的热门单曲。虽然他也可以亲自创作自己表演的歌曲，但他却选择去利用其他音乐制作人的作品。这使得他对整个娱乐世界产生了更大的影响。是的，借助于他的音乐和相关商品的销售，他变得非常富有。但许多人也会从他的这些财富中分得一杯羹。这就是真正开源环境中的协作所带来的积极影响，每个人都能从中受益。

猫王死了。愿他的灵魂永得不朽。

也许有人会说把滚石乐队、猫王与Eric Raymond的著作《大教堂与集市》（*The Cathedral and the Bazaar*）放在一起来做类比有点儿牵强。毕竟，猫王最近距离接触的集市可能也就是麦克斯韦大道的孟菲斯跳蚤市场（Memphis Flea Market）。而在大教堂，你最不可能找到的就是那位创作出*Goat's Head Soup*[②]专辑的人。因此，让我们来看看一个真实的大教堂，或者更确切地说，一种曾经流行于大教堂里的语言：拉丁语。

拉丁语曾经拥有过像今天英语一样的辉煌地位。随着罗马帝国逐步征服世界，从大约公元前250年直到6世纪，拉丁语的使用便一直在稳步地增长。在这段时间里，罗马天主教会宣称只能用拉丁语这一种语言撰写科学、哲学和宗教作品。因此从那时候起，如果想成为引人注目的牧师或学者，你就必须学会说拉丁语。然而，随着

① 福音（gospel）是美国黑人的一种宗教音乐。——编者注

② *Goat's Head Soup*是滚石乐队于1973年8月31日发行的一张专辑。——译者注

罗马帝国的逐渐衰落，外夷蛮族开始入侵，那些入侵者的好战本性使得他们并没有兴趣让自己显得智力超群，他们通常都是按自己喜好来修改拉丁语。他们加入了自己民族关于杀戮、伤害和掠夺的词语，因为他们认为上帝也赋予他们征服这门语言的权利。与此同时，祭司和学者却纷纷谴责这种行径玷污了他们心爱的拉丁语，并进一步向不断缩小的宗教人群宣传捍卫这门语言的重要性，直到它最后归隐在宁静的大教堂里，只有大教堂里的人们才知道如何使用拉丁语。

另一方面，英语逐渐成为一种国际化语言，这并不是因为它的纯粹性或神圣性，而是因为它的适应性。它兼容并包，几乎所有的外来词或概念都能纳入日常的使用。迄今为止，它比世界上任何一门其他的语言都更具备与不同文化对接的能力，而且它做的事情就是连接。在这种我们称之为“生活”的集市中，集合了各种自然多样性，其中采用英语的国家是最多的，这主要归功于它连接万物的能力。

这也正是OSS（开源软件）能够得以蓬勃发展的原因。任何人都能以任意的方式将它们与各种事物连接在一起。这都是源自它的开放性：开放的标准、开放的协议、开放的文件格式和开放的一切。OSS世界里，没有什么事物能够隐藏。这种开放性使得软件开发人员能看到他们的前任是如何处理事情的。如果喜欢之前的做法，他们大可沿用，这很棒。他们因此也能找到有用的工具并节省宝贵时间。如果不喜欢，他们也大可挥洒自如地查看过去的错误并做改进。

同时，那些在“大教堂”里面忙着开发“高级”软件的开发人员最终却只能眼睁睁地看着他们的市场被开源世界的开发人员占据。为什么呢？因为这种教堂式的软件开发人员只会为同在“大教堂”里的人们开发软件。不管这个“大教堂”是一个公司、社区还是国家，可迟早这些软件必须要与大教堂之外的软件进行交互，否则就只有死路一条。

Unix哲学的拥护者，特别是今天的Linux开发人员敏锐地意识到这一点。例如，与Windows系统共享混合环境的Linux系统会运行Samba，也就是一种与Windows SMB网络服务兼容的文件共享应用程序。具有讽刺意味的是，根据基准测试的结果，Linux的表现实际上要优于运行Windows SMB网络服务的微软Windows 2000系统[①]。

大多数Linux发行版都能够访问位于其他磁盘分区上的Windows文件系统。那

① 请参考John Terpstra的论文“Using the SNIA CIFS Benchmark Client to Test Samba Performance”（使用SNIA CIFS基准客户端测试Samba的性能），它被收入2002年加利福尼亚州圣克拉拉召开的CIFS会议会刊中。

反过来呢？Windows能够访问位于其他分区的Linux文件系统吗？Windows却做不到这一点。作为闭源的专有系统，它就像一只常年将头埋在沙子里的鸵鸟一样，假装世界上根本就不存在其他的文件系统。

到现在为止，你可能感受到了OSS和遵循Unix哲学的软件之间的一些共通之处。与那些大型“企业级”（像一些人所认为的）闭源的专用软件相比，OSS的许多作品看起来确实只是些“小儿科”。毕竟，这些作品通常都是由一些个人独自开发的，至少在它们的初期阶段是这样。因此，在早期这些作品只是一些典型的小项目，它们虽然缺乏利于市场营销的闪光点，却拥有扎扎实实的功能。不过没关系。请记住在Unix的世界里小即是美。

Eric Raymond称这些作品为“挠到程序员痒处”的软件。这似乎是许多OSS开发人员的共同目标，他们只是希望编写出他们喜欢的程序。因为日常工作压力，他们往往是“背水一战”，所以没有太多时间去完成那些花里胡哨的功能。因此，他们通常都会忽略掉那些营销人员最爱的、华而不实的“闪光点”，他们注重的是只做好一件事情，这便意味着他们的作品是精干实用的应用程序，并确实能够解决个人的需求。在那些成功的案例中，这些最基本最行之有效的解决方案可以激发人们的想象力。这正是热门OSS应用程序诞生的典型过程。

一旦点燃了想象力的火花，OSS开发社区的其他成员就会开始行动，来为这个方案贡献自己的力量。起初，这个程序会吸引为数不多却很忠实的追随者。然后，该软件最终焕发蓬勃的生机，开始超越它最初的设想。在此期间，数十名程序员会把这个“第一个系统”演变成为一个更大更具包容性的“第二个系统”。在撰写本文时，Linux本身就处于“第二个系统”的阶段。它正在享受自己的快乐时光。

显然，如果有人想将这种个人独自开发的OSS项目发扬光大，成为风靡一时的软件热潮，那他就必须尽早建立一个原型。仅仅对软件的伟大想法夸夸其谈是远远不够的。在开源的世界里，最受人们尊敬的是那些编写软件最初版本的原创者，以及在后来作出重大贡献的人士[①]。

如果想要吸引尽可能多的用户，OSS项目就必须提供适用于多平台的解决方案。因此好的OSS开发人员就会舍高效率而取可移植性。利用特定计算机上特定硬

① 这并非是要抹杀那些在开源社区起到催化作用的有识之士的功劳。好的想法必然来自某个地方，而这其中的一些人是名副其实的创造力源泉。

件功能优势的软件走不了太远，追随者充其量也不过是那些使用着相同硬件平台的人们。通过从封闭式专有架构走向开放的架构，程序就可以尽可能地出现在更多的平台上，从而最大限度地拓展自己的市场潜力。

让我们来看一下微软的IE浏览器和它在开源世界的竞争对手Mozilla。在撰写本文时，IE占据的市场份额要高于Mozilla，最主要的原因在于它是微软Windows平台上的主导浏览器，并且Windows目前占据着台式机市场的最大份额。另一方面，Mozilla能够运行在Windows和几乎所有的Linux和Unix平台上。随着用户开始冷静地对待IE的市场炒作，他们慢慢会发现Mozilla是一个非常好用的浏览器，越来越多的人开始选用Mozilla。与此同时，在除Windows之外的平台上，IE浏览器的安装用户群并没有什么大的增长，因为它缺乏可移植性，人们很难轻松地将它移植到其他的系统。如果Mozilla能够成功地吸引到一半安装Windows的用户群，那它就能赢得这场比赛。任何能够获得一半Windows用户群青睐的可移植浏览器，都会吸引到Linux用户群中的大多数人。[①]

为了使OSS和接口能够方便地移植到其他平台，采用纯文本文件来存储数据将大有裨益。这种开放性使得其他人可以查看这些文件格式，并使得人们可以就其中内容而展开有益的辩论。这场辩论最终会形成一种能够为数据传输和存储所用的开放标准，采用它们的广泛程度要远超闭源的专有标准。

OpenOffice.org团队的做法是很能体现这种思路的绝佳例子。他们的文件格式采用了XML，这是一个很好的选择。XML是HTML格式的扩展集，它是一种Web浏览器使用的语言，用来显示文字、表格等内容。XML文件算是一种纯文本文件，因为人们可以用如vi或Windows记事本的普通文本编辑器（而不是文字处理软件）来编辑它。作为一个行业标准，它也许是结构化交互式操作数据的最佳选择，不单能作为纯文本文件来使用，也能为纯网络消息所用。毫无疑问，这个格式会随着时间的推移而进化。随着越来越多的人在关注这些数据文件，格式中的错误和遗漏将无处可藏。这会使得该文件格式得到加强和改进，比起任何现有的闭源文件存储格式，它的表现将会更好。

OpenOffice.org采用纯文本文件当然不是什么新鲜事。现实世界的办公环境里一直在使用纯文本。商业信函通常是由那些20磅的10号白色信封来传递的。这些档

① 参见http://www.planettribes.com/allyourbase/。

案具有以下特点。

- 便携性：信封可在办公大楼里或世界各地发送。可扩展机制能够将它们从一个地方转移到另一个地方，并在个体和群体之间传递。
- 易于获取：任何人都可以打开这个信封。你并不需要从专门的供应商那里花高价去购买特殊的专有工具来浏览里面的内容。
- 可供搜索：文件可以加上索引，收件人可以根据自己的需要将文件按照不同机制归类在不同的档案柜里。人们还能够采用随机存取或连续存取的方式得到这些文件。人们可以搜索全部文本内容，尽管速度会有些慢。
- 强大的适应性：在今天或是未来，文件被存储和检索的次数能够达到数千次。人们完全不需要去升级自己的办公处理方式来阅读最新的文件，或是安装旧版本以查看十年前的旧文档。

是的，我们并没有建议你放弃电脑，重新去使用纸张。纸张仍然是一份为数据开具的死亡证书，其中的理由我在本书其他地方提到过。然而，数据还面临另外一种死亡威胁：闭源的专有文件格式。当存储的数据采用的是由单一供应商所拥有而且是一种人类不可读的文件格式，那么这些数据就存在着巨大风险，正如秘鲁国会议员Edgar David Villanueva Nuñez博士在一封发送给微软的信件[①]中指出的那样。

专有系统和格式的使用都会使得整个国家更为依赖某些特定供应商。一旦国家确立了一个使用自由软件的政策（当然，使用自由软件还是需要付一些费用的），从一个系统迁移到另一个系统反而会变得更加简单，因为所有的数据都是采用开放格式来存储的。

开源软件需要开放的数据格式。

在OSS开发人员发布那些供公众使用的软件时，他们利用了软件的杠杆效应。这种杠杆效应是双向的。首先，开发人员可以在自己的作品中使用其他OSS项目的代码。这样可以节省整体的开发时间，并有助于降低开发成本。其次，发布给全世界的优质OSS通常也会吸引到其他开发人员，他们可以利用这个软件来完成自己的工作，其中一些作品最终可能会找到一种回归到原始作品的方式。“赠人玫瑰，手有余香”。[②]

① 参见http://www.opensource.org/docs/peru_and_ms.php。

② 路加福音6:38：“你们施舍，也必会获得施予。人家要用十足的、连按带摇，以致满溢的升斗，倒在你们的怀里。因为你们用什么量器量给人，人家也用什么量器量给你们自己。”

有一种做法能够增加OSS项目成功的几率：使用脚本来提高杠杆效应和可移植性。虽然软件世界里有很多核心的底层编码人员，可越来越多人没有时间或无法尽情地为底层工作。给一两个流行的脚本语言提供接口会显著增强软件的可用性。

如果这些接口足够流行的话，它们就会成为OSS社区里的重要子社区。这些子社区致力于让人们体验并拓展这些接口。他们定期召开用户组会议和研讨会，并且在网上邮件列表里发布文章，通常它会成为社区中心，基本上大多数围绕着这个流行接口而展开的活动都是由这个中心来召集。

开源社区里有两个这样的例子，Perl社区和CVS（Concurrent Versions System，一种版本控制系统）社区。在OSS社区内，CVS吸引了不少人。因为几乎所有的CVS命令都可以在脚本中使用，于是就出现了众多工具和插件，它们通过新颖有趣的方式与CVS命令相结合。CVS的原作者肯定没有预料到人们能够开发出这些奇妙的用途。

CVS特意避免了CUI的使用。只要传递的参数正确，每一条CVS命令都可以在命令行中运行。这催生出了更多的CVS插件。以基本CVS命令如checkout、update和commit为基础，构建那些为提高CVS性能而开发的集成工具。众多使用这些命令的插件通常都拥有CUI，但基本的CVS命令却没有。这正是CVS得以蓬勃发展的主要原因之一。

作为开源的脚本工具，Perl因晦涩难用而恶评如潮。它的功能比普通Unix脚本如Bourne和Korn Shell还要强大，非常善于与其他软件交互。几乎所有的Perl程序都能充当过滤器。事实上，它的全称Practical Extraction and Report Language（实用型摘录与报告语言）就恰恰说明了Perl的设计开发宗旨。当一个程序从某个地方提取、修改并报告信息（即它的“输出结果”）的时候，它充当的正是过滤器角色。

因为它的可扩展性，所以Perl的接受程度也在与日俱增。在这门极其不好用的语言中，黑客们（指的是这个词的褒义方面）仿佛找到了一片名副其实的乐土，他们尽情地添加代码库文件以让它适应各种各样的情况。今天，你可以找到能够处理各种事物的Perl模块，从复杂的数学内容、数据库开发到为万维网所用的CGI脚本。

这一点儿都不奇怪，可以说Perl是关于Unix哲学中那条次要准则“更坏就是更好”的绝佳例子——它完美地诠释了为什么一个不够好的应用（如Perl）反而比那些足够好的应用（如Smalltalk）有着更高的生存几率。几乎没人会否认Perl是一门

古怪的有着可怕语法的编程语言，而且谁也都没有想到Perl会大获成功。

读到这里，你应该不难发现，OSS往往遵循Unix哲学。OSS很适合这种早期Unix开发人员悉心呵护的“集市”开发模式，而今天的Linux开发人员也都早已全面接受这一风格。不过有别于Unix哲学更加侧重设计方法的特点，开源社区中很多令人信服的作品却表明他们更注重软件的开发过程。用一句老话来形容，Unix哲学与开源开发模式的结合有如天作之合，互相促进，相得益彰。

开源社区取得过另外一个重大进展的领域便是市场营销。在当今竞争激烈的计算世界中，推出拥有良好设计原则的高质量软件是不够的。开发者必须大声向全世界宣告自己的成就。“酒香不怕巷子深”行不通，如果无人知晓，就算产品再好也没什么用。它会逐渐消失或默默无闻地湮灭，就像是一件多年不见天日的珍宝那样。

由于缺乏扎实的营销策略，许多优秀的Unix程序已经消失得无影无踪。如果我记忆超群，我说不定能说出几个程序的名字。明白我的言外之意了吧？

微软也许比其他任何公司都更充分理解营销策略的价值所在。这也正是为什么这家公司一直都很成功的主要原因。微软已经推出了很多个热门应用，但它最强的地方在于有能力说服大家相信它的软件是这个星球上最好的产品，而不管实际的质量究竟如何。

对于微软的软件，我最不喜欢的一点就是，那是在一个封闭大教堂式的开发环境中产生的。微软拥有一些非常有才华的人，他们编写了一些很优秀的软件。但在全球的技术创新方面，微软并不是一枝独秀。这个世界上还有很多优秀的人才，他们不为微软工作，也同样编写了一些非常好的软件。因为它的大教堂开发风格，所以微软的开发人员基本都“与世隔绝”。他们无法充分利用软件的杠杆效应，因为没有任何可以和其他人分享的东西，反过来别人也无法改善他们的软件来作为分享的回报。因此，大多数Windows操作系统的创新工作都是由微软内部的开发人员来完成，于是也就被他们的开发视野所限制了。另一方面，开源社区的开发人员则可以从更大规模的开源社区去吸取好的想法。纯粹论规模的话，开源社区完胜微软这种单一的内部开发社区。尽管微软有着高超的营销技巧，可这改变不了它的衰败命运。

在我撰写本书时，Linux开发社区（据说它已经拥有了超过一百万的开发人员）已开始进军桌面环境领域，长期以来这被视作是微软的大本营，并且大家都认为

任何外部人员都难以逾越微软在这个方面的成就。可就在短短的18个月里，Gnome的开发人员（Linux上流行的一个类似于Windows的应用环境）却将Linux从一个只具有Unix命令行的用户界面平台转变成了与Window 95功能相当的环境。要知道微软可是花了五年多才完成了这种从MS-DOS到Windows 3.1再到Windows 95的转变。

我们无法想像微软如何才能跟上这种创新的速度，这样的表现简直无人能敌。KDE社区的规模甚至可能比Gnome社区的规模还要大，它早已证明自己相当有能力去创造出具备商业品质的桌面环境软件。除非微软能够重新改变自己的开发模式，否则它的Windows操作系统早晚会成为下一个OpenVMS。（OpenVMS是典型的优秀小团队开发优秀大型软件的例子。）

微软曾经漠视Linux和开放源代码社区近十年。然后，他们却突然开始嘲笑Linux，声称这只是“一个服务器系统”，并不适合作为桌面环境来使用。现在，微软已经孤注一掷地投入到桌面领域的战争，而KDE的拥趸们却占据了上风。

甘地是对的吗？这将由市场来决定。

在结束本章之前，我想讨论一个问题，这个问题自2001年9月11日美国世贸中心和五角大楼遭受恐怖分子袭击以来一直萦绕在每个人心目中，而且也是后现代生活和计算机世界的重要方面，那就是：安全。

自那一天开始，安全问题以这样或那样的方式驻留在每个人的脑海中。这种共通的安全意识也已经“植入”进计算机科学家的头脑。也许，Unix哲学和开源奇特地掌握着解决此问题的关键。

在信息安全和加密技术的世界里，有些人坚持认为想要提供最高级别的安全性，开发人员就必须隐藏一切，这是未雨绸缪的行为。早期Unix开发人员的做法却与此大相径庭。他们并没有隐藏用来加密的密码数据和算法，而是选择对每个人公开所有的信息。任何人都可以自由地尝试破解Unix的加密算法，而且还能够得到所有相关的工具。在共享的精神作用下，这些找到破解Unix口令文件的人会把他们的解决方案提交给最初的开发者，这些开发者就可以在未来版本中纳入这些加密算法的解决方案。多年来，人们反复地尝试去破解Unix的密码机制，其中有一些尝试颇为成功，这使得Unix逐渐成长为更安全的系统。如果没有别人的帮助，单靠最初的开发人员可达不到这样的效果。

同时，那些试图通过封闭源代码来确保安全性的系统却一败涂地。比如像微软这样将自己的源代码看得死死的公司，就只能对Windows NT系统的安全漏洞采取亡羊补牢的方式。

从某种意义上说，安全是一种数字游戏，闭源系统的胜算非常小。这个世界每出现一个恶意黑客（坏家伙），对应也会有大概100倍甚至1 000倍之多的白客。问题是，闭源公司只负担得起雇用其中10个白客来看管他们的专有源代码。而与此同时，在开源世界里，比坏人多1 000倍的白客却都可以查看源代码来解决安全问题。闭源的公司，不管有多大，愿意花多少钱，能够雇用的白客数量也远远比不上开源社区里白客的数量。

开源软件与闭源专有软件的比较中蕴含着深刻的道理。有这么多人盯着OSS的安全机制，OSS最终会证明这两个系统里面哪一个才是更为安全的系统。此外，随着个人、公司和国家越来越了解信息安全工作中对软件进行审核的重要性，很明显，唯一值得信赖的安全软件就是那款你拥有源代码的软件。

在某些时候，每个人都得自行决定是否要信任那些维护信息安全的软件生产商。对软件供应商，每个人都有着自己在道德和伦理上的批判标准。我们能够充分信任那个供应商吗？当你的财务状况、信誉、个体或国家安全受到威胁时，唯一可以接受的解决方案就是可以逐行验证软件的代码。

我们生活的世界里同时拥有开放和封闭的社会。封闭社会的运作模式就像是“大教堂”式的开发环境，源代码被层层封装，并且只能由少数拥有特权的人来决定哪些该留下、哪些该舍弃。在这样的社会里，新观念的传播极其缓慢。有效安全机制的开发节奏跟不上层出不穷的新威胁，新机制的开发周期可能需要数年。

这个世界的开放性社会看起来就像OSS社区一样杂乱无章，但里面也充满了各种天马行空的想象。开源社区的人们非常具有创造力，关于新技术尤其是安全技术的想法层出不穷。开放性社会能够迅速地提出创新想法，确保在短时间内解除各种威胁。就像Unix密码算法的开发人员一样，让系统处于易于攻击的状态反而最终会成就最高级别的安全水准。

第12章 Unix的美丽新世界

12

一张缺少乌托邦的世界地图不值一顾。

——奥斯卡·王尔德[①]

今天如果去听听大多数软件公司的营销口号，你就会发现每个公司都会号称自己掌握了通往软件“乌托邦”的不二法门：“我们的拥有集成系统解决方案”，“我们的流程能够加快开发周期”，“我们的图形用户界面可以让你更快上手”，“它能降低运行数据库应用程序的成本”。毫无疑问，这些公司确实花费了大量人力物力来合理地调整运作策略，他们的理由也非常具有说服力。为了传递这些信息，大部分公司因浪费纸张而消耗掉的树木资源可比我们为了建造房屋而用推土机推平的树木还要多。

我不是什么先知，但我确实粗略观察过这个现象。Unix操作系统及其哲学已经产生25年多了。在一个锐意进取、不断创新的行业，这样一种关于系统设计和软件架构的方法能够坚持这么久，可不是什么寻常事。它不单经受住了数千甚至数百万人暴风雨般的批评，它还继续蓬勃发展着，并越来越受到人们的喜爱。事实上，作为Unix的最新化身，Linux在不久的将来必定会成为全球最大的软件产业。

每隔大约10年，计算机世界就会有一次重要的范式转变[②]（paradigm shift）。早在20世纪60年代，随处可见的大型机主宰着计算机实验室。然后，在20世纪70年代小型机和分时系统促成了一个“第三代硬件”的黄金时代。20世纪80年代图形用户

① 奥斯卡·王尔德（Oscar Wilde），生于1854年，卒于1900年。英国剧作家、诗人、散文家。——编者注

② 范式转变指的是当现有的范式里面出现反常或不一致时，我们不能解决出现的问题，因此对现实的观点就要改变，同时也要改变我们感知、思考、和评价世界的方法，这种改变就叫作范式转变。——译者注

界面开始出现，采用计算机来处理个人事务的概念深入人心。在20世纪90年代，我们则见证了个人电脑商品化的过程，以及随着因特网和万维网而出现的全球性网络。新的千年，我们正在经历着软件资源大规模分散化、大规模定制，甚至是一种前所未有的行业协作化做法——开源。

有趣的是，Unix系统不仅是这些技术更新换代洪流中的一个部分，还是大部分变革的催化剂。从小型机开始，Unix就扩散到了整个计算世界。（有些人可能会说它就像癌细胞一样！）Unix上开发的X Window System是第一款引起轰动的图形应用平台。最早在Unix上运行的Apache Web服务器成为了因特网上最主流的Web服务器。而作为Unix在PC机上的一种实现而出现的Linux，吸引了一大群程序员来为之编写应用软件并免费提供给大家使用。

每一次范式转变都会给人们提供无数机会，总有一些个人和公司能够最先意识到这个趋势，且抓住机遇成为时代的弄潮儿。昔日Unix的追随者和今天Linux的倡导者曾经感受到并正在见证这些让世界翻天覆地的变革。

尽管Unix和其哲学的追随者一直都能在发生重大技术变革时处于领先地位，有些人还是声称Linux和Unix是与昔日小型机紧密相关的过时技术。他们说，现在我们面临的是一个勇敢新世界，这是图形用户界面的世界，这是网络化的世界，这是商品的世界[①]，这是一个无线的世界。

事实上，这是一个Unix的世界。因为无论发生什么事情，Unix都反复证明，它都拥有惊人的适应能力。当人们说采用除商业服务器之外系统的代价会过于昂贵时，Linus Torvalds的Linux[②]却证明他们错了。当人们说它无法在桌面领域与微软的Windows一争高下时，基于Linux的KDE开发者却认为这恰恰是一个挑战。今天，它已经成为我个人最喜爱的产品。尽管许多人认为Windows是比Linux更好的游戏平台，但我却想强调早在游戏先驱雅达利公司出售它第一个游戏主机之前，人们就在Unix上玩游戏了。

因此，技术竞赛还在继续进行，Unix哲学仍然将成为其中的推动力量，不只是

① 在我写这本书的同时，你花上200美金就可以从沃尔玛超市买到一台1.1 GHz主频运行某Linux版本的PC机。差不多的Windows机器要贵一些。当然，在读到本书的时候，你可能会觉得这个价钱贵了。没关系，就像我之前说过的，明年的机器会跑的更快，而且更便宜！

② 人们曾经试图干脆将Unix更名为Linux，因为市场觉得在后现代的计算机时代，Unix这个名字显得过于陈旧。不过每个人都知道，Linux只不过是Unix哲学的最新实现方式。

关于操作系统设计这一方面，而且还包括其他创新领域。在这一章中，我们要看看Unix哲学如何渗透到其他的项目、技术、方法和设计理论。许多技术都号称自己能引领软件进入佳境或是软件“乌托邦”。让他们吹吧，没关系，如果他们的哗众取宠能够引起别人的注意。至少这让我们知道了他们的存在。在人们的努力尝试中还是有很多不错的想法。我唯一的遗憾就是，我不能参与所有的这些尝试，不管是因为它们已经达成的还是承诺要取得的目标，每一个都对我有着莫大的吸引力。

为了更好地识别Unix哲学中哪些准则适用于哪些领域，表12-1列出了本章其余部分使用的缩略语清单。主要的准则是大写，其他那些较小的准则是小写。

表12-1　本章中用到的缩略语

缩　略　语	描　　述
SMALL	小即是美
1THING	让每一个程序只做好一件事情
PROTO	尽快建立原型
PORT	舍高效率而取可移植性
FLAT	使用纯文本文件来存储数据
REUSE	充分利用软件的杠杆效应
SCRIPT	使用shell脚本来提高杠杆效应和可移植性
NOCUI	避免那些强制性的用户界面
FILTER	让每一个程序都成为过滤器
custom	允许用户定制环境
kernel	尽量使操作系统的内核小而轻巧
lcase	使用小写字母并尽量简短
trees	保护树木
silence	沉默是金
parallel	并行思考
sum	各部分之和大于整体的效果
90cent	寻找90%的解决方案
worse	更坏就是更好
hier	层次化思考

下面要讨论的技术大多都拥有一个重要主题或是主导性概念，在开始具体讨论时我会重点标注。

Java

采用的准则：PORT、FLAT、REUSE、trees、 parallel和hier

核心概念：舍高效率而取可移植性

Sun公司的Java编程语言凭借自身优势，已经成为了计算机世界的新热点。之前，C语言向来是可移植编程语言的不二之选，如今这个地位正被Java慢慢地取代。在新应用程序的开发工作中，Java早已牢牢确立了其作为实践标准的地位。它并非十全十美，也有着自己的毛病。然而，广大应用程序开发人群正在迅速地采用这门语言。

Sun公司的推广口号“一次编写，随处运行”（write once, run anywhere）可谓是响亮地点出了它在可移植性方面的价值。虽然在可移植性上，Java比C要更胜一筹，但它的工作效率却不过是C语言的95%。然而，基于Java在可移植性上占据的绝对优势，我们很难理解为什么会有人在乎这5%效率上的损失。

问题仍然是积习难改。对于那些坚持要将硬件性能发挥到极致的纯粹主义者来说，C语言依然是他们的最爱。许多Linux开发人员，特别是那些Linux内核开发人士都是C语言的拥趸。其实，相对而言真正与内核打交道的人只是极少数，大部分软件开发人员都是在开发能够运行于Linux和Windows平台之上的应用程序。对他们来说，Java可以很好地满足需求。

在促进Java成为开放性标准的过程中，Sun公司游走于专有软件的自控性和开源的灵活性之间，取得的成果还算是中规中矩。很多开发人员都在质疑Sun公司的动机，为什么它依然强有力地控制着Java的所有权并顽固地把持着它的命运，他们认为Java的发展应该由推行开放式标准的机构来管理。不过，这些问题并不在本书的讨论范畴之内，且将它们留给营销专家和战略专家吧，我们侧重考量的是编程语言方面的技术和哲学。

代码重用很受Java程序员的喜爱，这主要是由于它面向对象的架构。通过分层继承机制，大多数Java类在继承链中利用了他人编写的代码。新的类往往是现有类的扩展。尽管Sun公司的营销策略并不那么尽如人意，可是代码重用的方便性还是使得Java迅速广泛地被人们接受。

Java遵循Unix哲学中的这一条准则——采用纯文本文件来存储数据，并结合了

它自己的专有文件机制。相比起一些应用程序环境仍然在采用二进制文件来保存配置信息，Java专有文件表面上还是具有可读性的。系统上的文本搜索引擎也可以为它们做索引，便于开发人员随时定位应用程序环境中的资料。

在并行操作领域，Java采取了实质性的举措。它的核心类包括一组用于创建和操纵线程的对象集。因此，今天很多用Java编写的应用程序都是多线程的。这并不总是一件好事情，因为有些开发人员已经发现人们在很多方面滥用了这一机制，但大体上Java社区还是对这个机制颇为满意。

Sun公司另一个正确的做法就是Javadoc。这个工具可以用源代码的注释来产生一份HTML格式的API文档。由于它的输出格式是HTML，人们大多会使用网页浏览器来查看Javadoc文档，而不是将它们打印成纸质文件，因此，也就间接保护了森林资源。

Java做对了这么多事情，那有什么是它做得不对的吗？当然有。C语言有一个可怕的`#include`指令机制，正是反复导致变量名冲突的罪魁祸首。Java采取的`classpath`（类路径）机制也犯了同样的错误，它简直就是在重复C语言之前的噩梦。一次又一次，我们看到程序员深陷在类路径问题的泥沼，就像C程序员使用`#include`指令碰到的窘境一样。也许某一天会有人能够发明解决这一问题的方法。在此期间，我们只能这样将就。

最后，关于谁该主宰Java命运的问题也会时不时浮出水面。许多铁杆Linux开发人员习惯性地避免使用Java，因为他们觉得这不是他们设想的那种开源软件。这个想法可能过于死板了。Sun公司作为一个仁慈的“独裁者”，对Java还是颇为尊重。它给所有想要拥有Java的人提供了源代码，并能够回应大家的建议和批评。Sun公司只是想保留一些控制权，以确保Java不受别人影响，或被其他实体夺走所有权。

这个所谓的其他实体其实指的就是微软公司。Sun与微软水火不容，它们都在奋力夺取控制软件开发环境的垄断地位。与此同时，微软一直试图通过.Net和它自己的C#编程语言来抢占Java的市场。Java的生存会受到这些冲击的影响吗？很有可能。不管微软如何炒作，与微软推出的任何产品相比，人们向来都更为认可Java的可移植性。大量开发人员坚信，可移植性是不可或缺的特性。

面向对象编程

采用的准则：SMALL、1THING、REUSE、hier
核心概念：充分利用软件的杠杆效应

在讨论完Java之后，你可能会疑惑为什么我们还会提到面向对象编程(object-oriented programming，OOP)。原因就是，虽然Java是今天最流行的面向对象语言，但它并不是OOP的全部。而且，令人意外的是，如果执意去做的话，一些程序员的做法表明人们还是可以用Java编写出非面向对象的程序。(我想要补充一点，这些人得自行承担这么做的风险。)

OOP代表了一种远离类似于C和Pascal那种结构化或过程式程序设计的做法。在计算机领域，这样的巨变平均每隔10～15年就会发生一次。发生巨变时，它会彻底地改变我们看待一切事物的视角。这种巨变将驱动着无数软件去实施新的模式。通常，它会激发人们去重新编写那些旧软件，在新的架构下这些软件改颜换貌，其中会包含很多有趣的闪光点。

OOP的概念非常符合Unix哲学的理念。过程式语言中的一些函数往往要消耗一个内存页甚至更多的资源，而采用OOP风格精心编写的对象所消耗的资源却少很多。OOP是“小即是美”这一准则的完美体现。事实上，那些最能掌握Unix哲学精髓的程序员才会成为最好的OOP开发人员，他们很明白小工具更易于交互的内在价值。

然而，它也有缺点，人们不难想象这样一个可怕的情景，我们拥有数量繁多的重用类，其中每一个类都是由患有NIH综合症的程序员编写的，每一个类都是某个现有类的延伸。这样的情景在OOP环境中迅速涌现，因为OOP正处于“第二个系统”的发展阶段，几乎每个人都在这个代码乐园里尽情地编写着面向对象的代码。人们只能祈祷OOP能够尽快度过这“第二个系统”的发展阶段。

从表面上看，良好的OOP系统包括了大型的小对象集合，每一个对象都有自己的单一性功能意图，并具有层次化和高度集成的组成结构。这就是OOP的本质。如果构建良好，这种大型系统能够拥有高效性能。就像昔日的过程化架构一样，人们也有可能滥用这种面向对象的概念，而且这一系列对象的关系图可能看起来并不太像一棵树，而更像一碗意大利面条。有时候，人们希望程序员只构建一个面向对象

的“小屋”，可他们却给你建造了一座面向对象的“宫殿”。重构可以解决这个问题，它是对软件内部代码结构和对象关系重新进行分析的过程，目标是降低软件的复杂性。至少，在下一个结构性转变之前，OOP依然会存在。

极限编程

采用的准则：SMALL、1THING、PROTO、custom、sum、90cent
核心概念：尽快建立原型，从而与客户一起开展迭代开发工作

如果去看一看http://www.extremeprogramming.org/what.html网页上关于XP（extreme programming，极限编程）的定义，你可能会认为，XP的创始人抄袭了Unix哲学的概念，给它打上新的标签并宣称这是自己的原创思想。这样想也没什么错。甚至可以说，XP的方法论不过是肯定了昨日Unix开发人员和今日Linux开发人员反复强调的一件事情：建立原型。现在就去做，不要再等待。

XP社区有一些值得称道的做法，在正式确定迭代开发过程的形式并将其价值宣导给这个世界的非技术性管理层和经理人方面，它做了很多的工作。XP沿用了许多Unix程序员反复强调的尽快建立原型的做法，并将其改头换面，令会计和营销人员也颇为满意。XP社区正忙着教会这些人，最好的软件开发方法也许就是坐下来尝试一下，然后和程序员聊一聊哪些是你喜欢的，哪些是你不喜欢的。这简直就是太棒了！

XP思想家们明确定义的一个重点领域就是测试。对于那些喜欢充当试验小白鼠的客户，Unix的追随者通常会为他们发布早期原型来测试自己的软件。虽然这个做法在Unix社区里取得了不错的效果，因为这样的话就等于拥有了很多测试人员，但偶尔它也会产生偏差。对此，XP的做法却是先写好测试软件。它声称这可以帮助你早一点明确软件的业务需求。

在实践过程中，XP方法论的早期测试做法可谓是有利有弊。在测试初期就让客户参与进来听起来是不错，但有时候，开发人员这么早就编写测试程序会使他们分心，从而无法集中精力应对客户的要求。更糟的是，随着时间的推移，需求会发生变化，人们经常要返工并重新测试。这样的话，在软件开发项目快要完成的疯狂赶工阶段，人们往往会完不成测试工作。当人们将这种方法推行到极限状态时，它会假定开发人员已经提前充分了解客户的需求，因为他们需要去编写测试这些需求的代码。

问题是，当需求发生变化时，人们没那么多时间来修改这些测试程序。所以，我们不如先不要理会测试这个方面，而是花点儿时间尽快建立原型，将它交付到用户的手中，然后再编写测试的代码。这样，我们就不会在编写那些不必要的测试代码上浪费时间，从而可以花更多的时间与客户沟通交流，了解他们真正的需求。其次，为了提高应用程序的总体质量，我们应该针对那些最有价值的功能去编写测试代码。这就像先使用代码分析器确定哪个程序最耗时，然后付出10%的努力来争取90%的改善一样，是一种事半功倍的做法。

尽管存在着这些分歧，我还是觉得Unix社区和XP社区相处得颇为融洽。他们有很多共同的目标，做事方法也很像。XP和Unix的关系就像是英国和美国。双方有着共同的根源，而且说的也是同一种语言。但是，英国人比较刻板一点，而美国人的处事方式则多了一点儿不羁的“牛仔”风格。

重构

采用的准则：SMALL、1THING、PROTO、FILTER、90cent

核心概念：简化再简化

近来，重构的关注点是该如何将程序员、黑客以及其他计算机程序修补匠们喜欢做的一件事情形式化：调整程序代码以获得更高性能。它的首要支持者是*Refactoring: Improving the Design of Existing Code*[①]一书合著者Martin Fowler。所有重构工作背后的基本思想就是，我们可以在不改变软件外部行为的前提下，去改善软件系统的内部结构。

重构做得不错的一点就是帮你认识到接口的重要性，也许在过去你并没有体会到这一点。在讨论一个由大量小工具集合而成的系统时，这一点显得尤为关键。除非已经严格定义好它们之间的接口，否则你最后的结果可能就是随机的、完全没有逻辑性的接口大集合。换句话说，一个烂摊子。

今天，开发人员和思想家撰写的关于在软件重构中应用设计模式和其他技术的书不计其数。唉，人们终于开始将这些在程序员脑海中萦绕多年的关于开发过程的想法形成文字。这类分析能够让我们更加了解我们做了些什么，从而可以采取行动来改善我们做过的工作。

① 本书中文版名为《重构：改善既有代码的设计》，已由人民邮电出版社于2010年出版。——编者注

关于重构的一条公理就是，应该采用一种小规模、渐进的模式来改进代码，而不是一股脑地推倒重来。在这一点上，Unix哲学家和重构者的做法很类似。Unix开发人员曾经采用的就是这样的小规模增量迭代的开发模式。然而，就像XP方法论与Unix哲学存在差异一样，Unix的做法虽然与重构者有很多的共同点，但至少在一个方面是与之背离的，即“推倒重来”的意愿。重构者几乎从来不提倡完全摒弃某个程序的做法。而Unix开发人员却经常构建“第一个系统”，把它扔掉，然后再从头开始建立“第二个系统”，因为他们知道这是前往效果最佳的“第三个系统”的便捷路径。因此，从这个方面来说，Unix开发人员表现得更为大胆激进。有时候，如果一个系统糟糕得无可救药，但人们却想要以最快速度实现软件“乌托邦”的话，就只能另辟蹊径。

人们在重构时做出的很多决定，大都是让这个软件符合特定的设计模式，或是当程序不适用任何现有模式时，去重新发明一种新的设计模式。所有的设计模式，不管看起来它多么地具有说服力，其实充其量也只是原本设计模式的一个超集。也就是说，每一个程序都是一个过滤器。就好像一句老话，一切程序都只是0和1。就算是最复杂的数值计算或逻辑系统，我们也总能将它简化为一系列二进制代码。

在重构的过程中，也就是寻求一个正确设计模式来作为自己的架构模型时，你需要记住，将程序的形式表现为一个只有单一输入和单一输出的过滤器之前，你可能还没有完全理解程序的逻辑过程。如果你已经具备了这样的认识，就可以着手准备重构了。

Apache Jakarta 项目

采用的准则：SMALL、1THING、PROTO 、REUSE 、sum

核心概念：这个开源合作项目充分利用了很多人的工作，并产生了极大效益

当看到Apache Jakarta 项目[①]体现出的很高的协作水准、对标准的忠实执行度、真诚的技术交流以及始终如一的高质量软件时，人们只能得出一个结论：开源开发模式是行得通的。

① 参见http://jakarta.apache.org。

该项目的管理方式、软件编写方法和用户对此项目广泛接受的程度都可以证明，Jakarta打好了一个坚实的基础，能使所有专有软件开发人员坐立不安并为之侧目。

是什么让这个项目如此成功？它的成功并不只是因为某一个独立因素。我们可以这么说，是所有因素集合在一起，使其成为了一个强大的、具有高度凝聚力的开发组织。但如果真让我们在里面挑出一个最有影响力的因素，那恐怕就是——社区比软件本身更重要。换句话说，很多人凝聚在一起开展的合作系统，比起那种开发人员为单一实体（通常是公司）而工作的封闭性系统，效果要好得多。

Jakarta是免费软件，任何人想要都可以得到。由于它公认的高品质，大型企业一直在使用它来运行自己的关键性业务。由此产生的连带效应就是，涌现出了很多提供定制服务以适应特定行业需求的顾问和开发人员。这体现了一个重要原则：免费软件还是有钱可挣的。

我们已经看到，计算机行业有过好几拨充满新机会的浪潮。首先，是硬件风潮，大部分资金都投入在新CPU、内存和存储设备的开发工作上，从而使得我们积累了足够的力量进入下一个浪潮。大型公司如IBM和DEC都在这一拨热潮中赚得盆满钵溢。第二次是软件风潮，我们都见证到如Lotus、Novell和微软这样的公司在这拨热潮中拔得头筹，从满足公众对软件和应用的商务需求中获取巨额利润。接下来的一拨风潮是服务，在只要人们想要就能获得所需内容的时候，最终用户已经不再关心软件内在的功能是什么了。

请注意，当前两波浪潮各自开始背离“价格-性能模型”的时候，也就意味着它们的终结。PC机的商品化带走了硬件的利润。同样，免费软件也使得人们靠销售软件几乎赚不到什么利润。现在的盈利点在软件服务端。当每个人都能够以一种几乎可以忽略不计的代价得到一份软件或硬件的时候，那么服务（主要是通过定制方式）就成为决定着高下的关键性因素。在撰写本书时，这种现象已经出现了，像红帽子网络（Red Hat Network）、CNN和《纽约时报》等媒体提供的各种额外付费服务。我们已经从工业时代的大规模生产格局进化到大规模定制的信息时代。

因此，请留心观察Apache Jakarta 项目的动态。它是一个缩影，向我们展示着整个开源开发社区里最美好的一面。

互联网

采用的准则：REUSE、trees 、sum 、worse 、hier

核心概念：如果更坏就是更好，那么充斥在互联网上的烂网页恰恰证明网络会持续存在

网络中的网页数量超过了3亿，这充分表明万维网是一个因人海效应而产生的庞然巨物。在推广人类提供的最好资料的同时，它也暴露出了人类腐化堕落的一面。每次在你以为已看穿了这一切时，网络总是会报复性地给你重重一击，让你觉得自己是在螳臂当车。从以往的记录来看，这可能会持续很长很长的一段时间。

如今，重用是网络上流行的概念。一方面，上百万人在网络上非法共享和重用受版权保护的音乐资料；另一方面，他们还在合法地分享自己喜欢的HTML代码片段、图像、图标和JavaScript程序，用来创建自己的网页。事实上，在代码重用方面，很多非专业人士也许反而比大多数软件开发人员做得更好。其实，我们并不在乎网络世界充斥着多少垃圾。关键是通过大多数网页浏览器提供的查看HTML和JavaScript源代码的功能，我们每个人都得以共享这些垃圾。互联网就好像是这个世界最大的垃圾箱，半个世界都被掩埋在垃圾箱底下。

当然，万维网上也有很多非常棒的内容。基于文本的搜索引擎总是能让你轻而易举地找到有价值的内容。只要认识到，互联网上乐于分享的人至少10倍于怀有私心的人，你就能适应这个网络世界。你甚至可以享受一下自己为互联网作贡献的乐趣。

也许，通过最终大大遏止从各地打印机中吐出的纸张数量，万维网才能够证明自身的价值。还记得当计算机出现的时候，大家都说这会引领无纸化的办公室吗？然而，出乎意料，人们发现，电脑的出现却反而促使大家使用了更多纸张。但是，在办公室的同事中分发纸张得到的乐趣，要远远少于将资料发布在互联网上，然后看看全球各地有哪些人会来观看的喜悦。这样一来，互联网就在实际层面上帮我们履行了Unix哲学的这条小原则——保护树木。

无线通信

采用的准则：SMALL、PORT 、custom 、worse

核心概念：廉价、高效、移动的通信可谓是下一个“杀手级应用”

比起固定电话，手机通话效果并不是那么理想，听筒时常会出现杂音。当你一边穿过高速通道一边与别人讨论重要事情的时候，通话往往会掉线。相比家中的固定无绳电话，它们的电池似乎需要不断充电。但我们都知道自己喜欢它们的理由：便携性。

手机也许是能说明为什么我们要舍高效率而取可移植性的最好例子。手机通话效果的保真度并不是最重要的，即便如此人们还是发现自己越来越频繁地使用手机，而不是家中的固定电话。那些不知道如何使用可编程微波炉的人们对于如何使用手机却非常在行，他们清楚该按哪个按钮表示确认。这并不是因为手机的使用比微波炉更简单，完全不是。原因只是人们想要沟通的意愿太过强烈，甚至超过了享受美食的意愿。

手机也将Unix哲学中那条小即是美的准则发挥到极致。我的第一部手机是个大块头，就像砖头一样重，就算在信号主要覆盖区之外它的表现也还中规中矩。我花在手机通话上的开销并不算少，然而，我花出的每一分钱都很值。如今购买一台超薄智能手机所需费用还没有在城市中过一个晚上的夜生活开销高，许多人还从无线服务授权的供应商代理那儿获得了免费手机。

一个更有趣的关于手机和其他无线通信设备的问题就是如何衔接。手机不只是用来通话的工具。我们希望所有家用电脑能做的事情手机也能做。这意味着它应该能够发送和接收实时视频、玩游戏、将口头讯息转成电子邮件、即时翻译国外的演讲、阅读电子邮件、上网和观看流媒体等等。

是的，手机也许无法像台式电脑一样去完成这些事情，但它们的表现应该满足我们基本的需求。这恰恰又再次体现出了更坏就是更好这一准则，也就是那些便宜而又有效的事物要好于那些又大又昂贵的东西。人们不会在意他们手机上的小屏幕比不上笔记本电脑上的那个大屏幕。他们需要的只是个表现不错而且易于携带的工具。

你怀疑手机完不成这些任务么？如果你只是在看……呃，不好意思，我需要接个电话。

Web 服务

采用的准则：SMALL、1THING、PORT 、FLAT 、REUSE 、filter 、sum

核心概念：在网络上实施Unix哲学

迟早有一天，大家都会意识到Gordon Bell关于“网络将成为计算机系统”的说法完全正确。[①] Sun公司后来借用这个说法，稍微修改了一下并作为营销战略口号——“网络就是计算机”。尽管在前进的道路上人们做了一些有趣的尝试来实现这一愿景（比如CORBA、DCOM），迄今为止还没有哪一个技术概念像Web服务一样，真正实现了“网络就是计算机”的这个承诺。[②]

Web服务使用的模型是一个小工具集合，里面包含着以网络为基础的元件或工具。简单地说，该技术总体上是以HTTP作为载体，在传输层上采用的是简单对象访问协议（Simple Object Access Protocol，SOAP）来作为通信语言。和你想的一样，它的结构与Unix哲学有很多相通之处。理想实现机制中每个Web服务都包含以下准则：

- 只做好一件事情；
- 作为客户端和服务器端之间双向消息传递的过滤器；
- 采用如XML这样纯文本格式来存储消息；
- 采用可移植语言编写的可重用组件。

大家是否会对Web服务变得如此流行而深感惊讶？不管是开源社区的DotGNU项目[③]还是微软的.NET计划[④]都是围绕这一技术而实施的开发框架。Web服务在商业应用中具有巨大潜能，它很有可能成为计算机世界下一个结构性转移的开端。

---

① 请参考Gordon Bell 1982年2月10日发表在《纽约时报》上的文章“Why Digital Is Committed to Ethernet for the Fifth Generation”（为什么DEC公司会致力于第五代以太网的开发）。网址是：http://research.microsoft.com/~gbell/Digital/Ethernet_announcement_820210c.pdf。

② 参见 http://www.w3.org/2002/ws。这个URL有时候也许会变更为http://www.w3.org/2003/ws，你可以先从上层网页http://www.w3.org开始，然后寻找相关的资料。

③ 参见http://www.dotgnu.org。

④ 参见 http://www.microsoft.com/net。

作为万维网联盟（World Wide Web Consortium，W3C）中的一个标准，尽管Web服务技术的进展非常缓慢，而它的对立派别却磨刀霍霍准备好迎接这一场长期血战。如果说Linux和Windows之间的对决就好比是计算机世界的末日之战，那么人们这种争抢Web服务空间的斗争就是个游击战。双方都理解交战规则，各方都显示出想要暗中颠覆对方的迹象。

在这场激辩中最大的争议始终是安全问题。微软的Passport登录机制已经遭受了AT&T实验室研究团队的猛烈抨击。[①]不出意料，微软的身份验证方法是从一个单一实体（即微软）那里去获取必要凭证。这引起了开源社区的愤怒，声称没有任何一个单一实体有权去掌控这样的凭证。

我们再次将这个问题归结为信任问题。在这方面，开源倡导者占据上风。人们无法完全信任由任何人类组织、实体或政府创造的安全机制，除非大家可以查看用于构建该机制的源代码。其他任何形式都将导致个人受到控制，并最终成为奴隶。

就算是大教堂里的僧侣也会谦恭地表示，他们并不赞同微软这样的做法。

人工智能

采用的准则：PROTO, hier

核心概念：如果能尽快建立原型的话，也许你就可以从它运行在电脑上的效果，了解一些事情

读完本章其他一些讨论，你可能会疑惑为什么我们会谈到人工智能（artificial intelligence，AI）这一主题。毕竟，应该只有那些年迈的终身教授和主流极客阵营里的极客爱好者才会喜欢研究AI吧？而且，关于AI的科幻小说不早已数不胜数了么？研究AI的群体长久以来已经告诉我们，总有一天电脑会比我们更聪明。其中大部分只被人们当作是技术狂想而一笑置之。所谓的“有一天”仍然还在将来的某个时候。AI社区从未真正提供过任何有价值的事物，充其量也不过就是像IBM的深蓝[②]那样打败一下国际象棋比赛冠军。

我在这里想强调的是，AI社区总会迎来它的春天，有可能比你想象的要来得快。

① 请参考David P. Kormann 和Aviel D. Rubin于2000年发表在*Computer Networks*杂志的文章“Risks of the Passport Single Signon Protocol”。

② 参见 http://www.research.ibm.com/deepblue。

你也许不那么喜欢它，因为当你和那些比你更聪明的人或事物打交道的时候，你可能意识不到这些人或事其实是受AI控制的。

首先，AI理论大多集中在该如何追求大规模地遍历决策树。这正是机器人在国际象棋比赛中与人对抗背后的基本思路。下棋的种种决策毕竟是有限的，尽管它包括的变数非常之多。那些老旧机器人棋手只能搜索为数不多的几层决策树。而IBM的深蓝使用了IBM SP2型号为RS6000的32位CPU，此外它还包括256个国际象棋加速芯片，从而使得它每秒钟能执行1亿次搜索，而且它搜索的层次深达30层之多。

还记得那个“下一……的硬件将会跑得更快”的想法么？是的，总有一天，我们的手表所拥有的能量都会与“深蓝”相当。我太期待了。然后，卡斯帕罗夫就可以随身携带“深蓝”[①]并弄清楚为什么他会成为它的手下败将。

其次，在开发人员尽早建立AI程序原型的时候，他们偶尔会发现程序本身就能告诉他们一些新的东西。AI软件开发便成为一个反复迭代的过程，它涉及的不只是最终用户，还包括了计算机自己。很明显，当人们越来越频繁采用这种方式与越来越快的机器和更聪明的方案交流时，这些程序就将变得更为智能。

当然，AI领域中这种扩大化的趋势也不是没有任何风险，长久以来科幻作家们也都是这么提醒我们。想要了解如何良好运用这项技术（至少从商业角度来说）的生动案例，请访问Amazon.com。你会留意到亚马逊网站说服你购买书籍的种种方式。它会列出：你购买过的书、你调研过的书、你想拥有的书、你的朋友们推荐的书、你妈妈买过的书、你拥有过却被前女友烧掉的书、你的配偶在你公文包里发现的书、你的狗吃掉的书、你在卫生间里读过的书以及你弟弟借走却再也没有还回来的书，等等。

这一种力求通过两个或多个不同的数据库来匹配个人行为模式的AI是一种相对较新的趋势。基于它拥有的巨大商业潜力，公司会运用它来获得最大化利益，它可能会在未来几年内成为热门技术。到目前为止，像亚马逊这样的公司正在探索数据库之间的明显关联性。然而，如果AI的启发式搜索方法能够开始为我们指出不那么明显的联系，它会更有趣。是的，亚马逊给你推荐一本关于在Linux上进行音乐创作的绝妙书籍时，这貌似很有意思。但在AI能帮助超市推算出该何时将你最喜欢

① 深蓝（Deep Blue）是由IBM开发的专门用以分析国际象棋的超级电脑。1997年5月曾击败国际象棋世界冠军卡斯帕罗夫。——译者注

喝的那款咖啡提价时，就没那么好玩了。

在这一章中，我们讨论了Unix哲学的思想是如何渗透到其他技术领域的一些例子。我们的讨论并不全面，如果我错过了你最喜欢的技术，那我深表歉意。今天我们拥有这么多令人目眩神迷的新想法，关于它们的讨论完全可以再写另外一本书。即便如此，哪怕有这样一本大部头书，我们的讨论也不可能完全，因为每天都有层出不穷的新技术涌现。

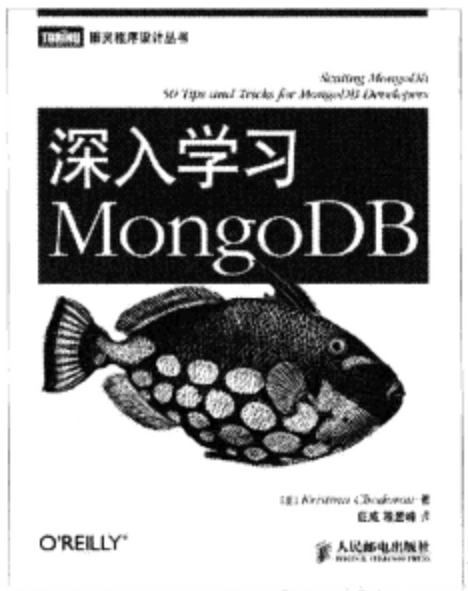
我希望你现在就可以在各方努力中找出那些类似于Unix的想法。当你看到对应模式慢慢形成的时候，就能更好地以兼容并蓄的态度拥抱它们，而不是采取一种封闭和排他的态度。一个共享想法的价值要高于封存于不同头脑中的两个想法。当人们分享他们的想法，而我们以开放的态度接受时，每个人都能从中受益。

本章也足以说明另一个重点：Unix众多版本更迭不已，但Unix哲学却经受住了时间的考验。对于Linux来说更是如此。Linux最有可能实现我们关于Unix的预测，也就是成为全世界的首选操作系统。可就算今天Linux通过它的开放性和包容性战胜了其他竞争对手和敌人，未来有一天Linux也可能会见证到自身被一个较新的操作系统所取代，以一种我们预想不到采用Unix哲学的方式。这其实没什么大不了的。当这种情况发生的时候，那些早就知道该如何“像Unix那样思想”的人们会为此作好准备。

让我们继续。做一件事情并把它做好。最后，你的辛苦付出将会获得更多回报。

TURING 图灵教育

图灵最新重点图书



深入学习 MongoDB
书号：978-7-115-27211-9
作者：Kristina Chodorow
译者：程显峰 巨成
定价：32.00 元

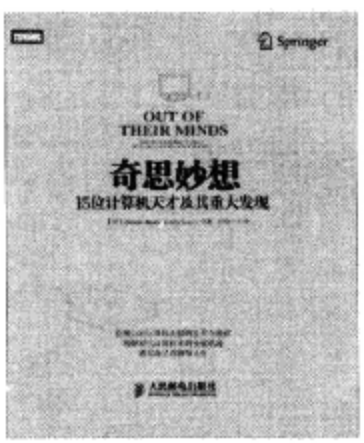
深入浅出的进阶学习手册

本书分两部分，分别来自 O'Reilly 的《MongoDB 扩展技术》与《MongoDB 开发技巧 50 例》两书。

前一部分“MongoDB 扩展技术”指导大家创建一个不断增长以满足应用程序需求的 MongoDB 集群，内容简明扼要，指导用户设置和使用集群存储大量数据并高效访问数据。此外，读者还可了解如何让应用程序兼容分布式数据库系统。

具体的主题有：

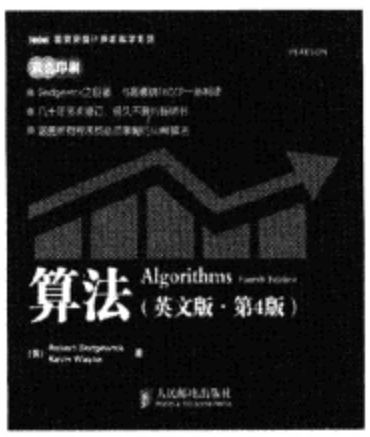
- 通过分片设置 MongoDB 集群；
- 在集群中查询和更新数据；
- 操作、监控和备份集群；
- 从程序设计角度，考虑如何应对分片、配置服务器或者 mongos 进程停止运行的情况。



奇思妙想：15 位计算机天才及其重大发现
书号：978-7-115-26881-5
作者：Dennis E. Shasha
Cathy A. Lazere
译者：向怡宁
定价：32.00 元

软件之道：软件开发争议问题剖析
书号：978-7-115-27044-3
作者：Andy Oram Greg Wilson
译者：张珉 鲍央舟 沈欢星
定价：89.00 元

计算机程序设计艺术，卷 4A：组合算法（一）（英文版）
书号：978-7-115-27050-4
作者：Donald E. Knuth
定价：129.00 元



算法（英文版 第 4 版）
书号：978-7-115-27146-4
作者：Robert Sedgewick
Kevin Wayne
定价：99.00 元

好学的 Objective-C
书号：978-7-115-27358-1
作者：Jiva DeVoe
译者：林本杰
定价：55.00 元

ActionScript 3.0 游戏编程（第 2 版）
书号：978-7-115-27289-8
作者：Gary Rosenzwig
译者：胡蓉 张东宁 朱栗华
定价：79.00 元